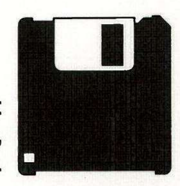
Markt&Technik

Peter Wollschlaeger

AMIGA ASSEMBLER-BUCH

Ein 68000er-Kurs mit vielen praxisgerechten Beispielen.

Mit ausführlichem Verzeichnis aller Systemroutinen und genauer Anleitung für das Einbinden von Assembler-Routinen in Amiga-BASIC.



Auf 3½"-Diskette enthalten:
Alle Beispiele im Quelltext, nützliche Utilities, wichtige Datenstrukturen und Programmrahmen.

Amiga Assembler-Buch

Der Autor:

PETER WOLLSCHLAEGER, Jahrgang 1939, gehört noch zu jener Technikergeneration, die digitale Schaltungen bis hin zum Computer mit einzelnen Transistoren entwickelte. Programmiert hat er damals nur nebenbei. Mit Einführung der Mikroprozessoren verlagerte sich seine Tätigkeit immer mehr von der Hardware zur Software. Durch ständige Weiterbildung, zahlreiche Kurse und Workshops bei den Herstellern sowie einige nachgeholte Semester in Informatik und über 20 Jahre praktische Erfahrung als Systemprogrammierer wurde er zum Experten für Mikrocomputer. Nebenberuflich arbeitet er als freier Autor für »Computer persönlich« und »mc«. Die journalistische Erfahrung aus über 150 Artikeln, immer unter der Prämisse geschrieben, schwierige Themen gut verständlich und angenehm lesbar darzustellen, steckt in diesem Buch. Bisher erschienene Bücher bei Markt&Technik: Atari ST, Assembler-Buch, Bestell-Nr. 90467; Atari ST, Programmierpraxis ST PASCAL Plus, Bestell-Nr. 90490

Jede höhere Programmiersprache wie BASIC oder Pascal, aber auch noch C, legt dem Anwender Beschränkungen auf. Sei es, daß bestimmte Dinge sich partout nicht realisieren lassen, oder sei es, daß Programme viel zu langsam laufen. Der logische Entschluß, dann tiefer einzusteigen, erhält auch prompt einen Dämpfer. Was nun käme, sei Assembler, und das sei wirklich nur etwas für Profis. Erst recht gelte dies für so einen komplizierten Prozessor wie den 68000.
Mit diesem Buch soll bewiesen werden, daß in Assembler auch nur mit Wasser gekocht wird. Es ist nämlich ganz einfach. Wer eine Programmiersprache gelernt hat und sie erfolgreich anwendet, der lernt auch andere. Assembler macht da keine Ausnahme.

Allerdings setzt Assembler ein gehöriges Maß an Grundwissen über computerinterne Dinge voraus. Aber keine Angst – nach einem Minimum an Theorie geht es sofort in die Praxis. Assembler-Befehle und DOS-Funktionen werden anhand kleiner Programme erklärt. Die Programme werden ständig schwieriger, das erforderliche Wissen wird dabei von Fall zu Fall mitgeliefert. Sie lernen schrittweise jede Menge über die Internas des Amiga-Betriebssystems, von »Intuition« bis hin zu Multitasking.

Aus dem Inhalt:
- Grundlagen des 68000-Prozessors
- Systemprogrammierung anhand vieler Beispiele
- Programmierung von Intuition
- Schnelle Grafik in Farbe
- Alle Systemroutinen mit Parametern
- EXEC und Multitasking

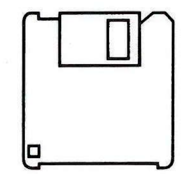
Die Begleitdiskette:
Sie enthält alle Beispiele im Quelltext, nützliche Utilities, wichtige Datenstrukturen und Programmrahmen. Die Programme sind mit dem Hi-Soft-Assembler »DevPac Amiga«, Verlag Markt&Technik, Bestell-Nr. 51656, geschrieben.

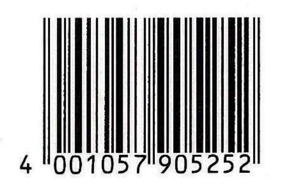
Hardware-Anforderungen:
Amiga 500, 1000 oder 2000 mit Diskettenlaufwerk und Monitor

Software-Anforderungen:
Assembler-System für Amiga (z.B. DevPac, Metacomco oder Seka)

ISB N 3-89090-525-0







DM 59,-
sFr 54,30
öS 460,20



Peter Wollschlaeger

AMIGA Assembler-Buch

Ein 68000er-Kurs mit vielen praxisgerechten Beispielen.

Mit ausführlichem Verzeichnis aller Systemroutinen und genauer Anleitung für das Einbinden von Assembler-Routinen in Amiga-BASIC.

Markt&Technik Verlag AG

CIP-Kurztitelaufnahme der Deutschen Bibliothek

Wollschlaeger, Peter
AMIGA-Assembler-Buch : e. 68000er-Kurs mit vielen praxisgerechten Beispielen ;
mit ausführl. Verz. aller Systemroutinen u. genauer Anleitung für d. Einbinden von
Assembler-Routinen in Amiga-BASIC / Peter Wollschlaeger. –
Haar bei München : Markt-u.-Technik-Verl., 1987. – & 1 Diskette
ISBN 3-89090-525-0

15 14 13 12 11 10 9 8 7 6 5 4 3
90 89 88

ISBN 3-89090-525-0

Einbandgestaltung: Grafikdesign Heinz Rauner
Druck: Jantsch, Günzburg
Printed in Germany

Inhaltsverzeichnis

Vorwort

Ich selbst habe seinerzeit Assembler auf einer IBM-360 gelernt. Damals war Rechenzeit noch unheimlich teuer, weshalb man uns Anfänger zuerst mit sehr, sehr viel Theorie traktiert hat, bevor wir den kostbaren Computer mit unseren ersten einfachen Programmen belästigen durften. Während der vielen Stunden Theorie hatte ich immer Probleme mir vorzustellen, wofür denn wohl was gut sein könnte. Richtig kapiert habe ich einiges von dem erst viel später, nämlich in der Praxis. Diese Vorgehensweise trifft man leider auch noch in vielen Büchern an. Nach zum Teil Hunderten von Seiten nackter Theorie kommt da endlich mal das erste Programm, wenn Sie Pech haben, sogar erst in Band 2.

Ich möchte da anders vorgehen.

Die Theorie soll nur soweit gehen, wie es für das Verständnis des ersten einfachen Programms unbedingt erforderlich ist. Leider sind wir dann schon beim vierten Kapitel angekommen, aber ganz ohne Grundlagen geht es halt doch nicht. Wenn das erste Programm läuft, lesen Sie weiter Theorie, bis hin zum nächsten Programm, das dann schon etwas schwieriger ist. So arbeiten wir uns langsam hoch, bis Sie zum Schluß in der Lage sind, auch komplizierte Programme selbst zu schreiben.

Noch etwas: Ein Assembler ist immer an eine ganz bestimmte CPU gebunden, hier den 68000. Diese CPU gibt es zwar im Atari ST oder im Macintosh auch, deshalb läuft aber ein Atari-Programm noch lange nicht auf dem Amiga.

Deshalb verzichte ich auf den ganz großen Leserkreis und schreibe dieses Buch speziell für den Amiga. Trotzdem, wenn Sie mal auf einen anderen 68000er umsteigen, dann können Sie Ihr hier erworbenes Wissen mitnehmen, Sie müssen nur die Internas des Betriebssystems des anderen neu lernen. Womit ich noch eine Aufgabe erwähnt hätte. Assembler-Programmierung ohne solide Kenntnisse von Exec, DOS oder Intuition ist nicht möglich. Auch das ist also nicht nur ein Kapitel in diesem Buch, sondern ein Faden (oder ein Seil), das sich durch alle Kapitel zieht.

Zum Schluß ein guter Rat: Wenn ein Programm nicht läuft, dann sind immer die berühmten Kleinigkeiten die Ursache. Hübsch häßlich ist nun leider die Tatsache, daß in Assembler jeder noch so kleine Fehler mit kräftigen Strafzeiten geahndet wird, weil nach der Fehlerbeseitigung kein einfaches RUN reicht, sondern einige Programme abgearbeitet werden müssen.

Lassen Sie sich dadurch nicht entmutigen! Mit jedem Fehler lernen Sie dazu, und schließlich machen Sie immer weniger Fehler. Eine gewisse Zähigkeit gehört allerdings dazu. Andererseits: Wer eine Programmiersprache gelernt hat, und ich unterstelle, BASIC oder etwas anderes können Sie schon, der lernt auch eine zweite Sprache. Generell ist Assembler nicht schwieriger als BASIC, nur leider etwas umständlicher. Zu diesen Umständlichkeiten gehört auch, daß Sie - im Gegensatz zu BASIC - hier wissen müssen, wie der Computer funktioniert.

Aber irgendwie macht es mir auch mehr Spaß, wenn ich meinen Computer sozusagen direkt programmieren kann. In einer Hochsprache bin ich immer auf die Güte (auch im Sinn von Gnädigkeit) des Compilers oder Interpreters angewiesen. Wenn das Ding ungenau rechnet oder viel zu langsam ist, ist mit dieser Erkenntnis die Sache ungelöst beendet, es sei denn, man kann in Assembler das Problem nun richtig angehen.

In diesem Sinn (und nicht aufgeben!!)

Ihr

Peter Wollschlaeger

Wer sollte dieses Buch wie lesen?

Das Buch wendet sich an Einsteiger und Umsteiger. Letztere können im Kapitel 1 die Abschnitte 1.1 bis 1.7 überspringen, im Kapitel 2 die Abschnitte 2.1 und 2.2, und das war's auch schon.

Ich muß die Umsteiger, soweit sie von den »8-Bittern« kommen, leider enttäuschen. Vorkenntnisse vom Z80, 8088 oder 6502 her nützen beim 68000 herzlich wenig, noch schlimmer, sie könnten sogar stören. Ernsthaft: Als Z80- oder 6502-Programmierer gewöhnt man sich gewisse Techniken und Denkweisen an, die zwar auf den 68000 übertragbar sind, aber dann nur unnötig lange Programme ergeben. Vergessen Sie alle Adressierungsarten, die Sie da gelernt haben, den Begriff Akku streichen Sie ganz, Banking und Paging erst recht und noch vieles mehr. Am besten, Sie vergessen alles!

Haben Sie schon 68000-Erfahrung, dann können Sie bei Kapitel 4 beginnen. Ähnliches gilt für die Leser, die von den Minis her kommen, speziell von der VAX.

Der 68000 ist zwar ein Prozessor, über dessen fantastische Eigenschaften man bücherfüllend hochgestochene Traktate für Diplom-Informatiker verfassen kann, aber genau das tue ich nicht. In den Kapiteln 1 bis 3 bringe ich nur die Grundlagen, die Sie kennen sollten, um die ersten Programme schreiben zu können. Dann folgt Praxis, Praxis, Praxis. Erst in den Kapiteln 10 und 11 beginnt die Würdigung des 68000 und dann folgt schon wieder Praxis.

Während des Praktikums wird folgende Linie verfolgt:

1. Schilderung der Aufgabe
2. Vorstellung der dafür erforderlichen Befehle und TOS-Funktionen
3. Das Programmlisting
4. Die Erklärung des Listings

Stellenweise wird diese Ordnung durchbrochen, weil es manchmal sinnvoller ist, den Punkt 2 im Zusammenhang mit dem Listing zu erklären.

Auf jeden Fall sollten Sie niemals zuerst das Listing lesen, sondern es im ersten Ansatz überspringen.

Die Fülle der Informationen aller Kapitel sich zu merken, dürfte schwierig sein. Deshalb sind im Anhang unter anderem die 68000-Befehle und die DOS-Funktionen in

einem kompakten Format zusammengefaßt. Darüber und mittels des Stichwortverzeichnisses sollte ein schnelles Nachschlagen möglich sein.

Besonders die Anwender des SEKA-Assemblers seien auf das Kapitel 11 und die Anhänge hingewiesen. Dort finden Sie LVO-Tabellen und andere wichtige Daten, die sich andere Assembler von der Diskette holen.

Falls Sie noch nicht mit dem CLI vertraut sind, lesen Sie bitte unbedingt den Anhang 5 noch vor dem Kapitel 4!

Kapitel 1

Assembler:

Was ist Assembler?

Wie programmiert man in Assembler?

Wann braucht man Assembler?

Was braucht man an Software ?

In diesem Kapitel soll zuerst einmal gezeigt werden, was Assembler ist, wann man ihn braucht und was man an Software benötigt, um ein Programm in Assembler erstellen zu können.

Eines vorab: Wenn mal ein paar Fachausdrücke auftauchen, die Sie nicht verstehen, einfach weiterlesen. Wenn wir sie wirklich brauchen, werden sie auch erklärt.

1.1 Ganz unten: Maschinensprache

Ein Computer an sich ist sehr dumm, nur sagenhaft fleißig. Gehässige Leute sagen auch, nur wer so dumm ist, ist auch so fleißig. Tatsächlich kann diese Maschine nicht bis Drei zählen, noch nicht einmal bis Zwei, sie kennt gerade die Null und die Eins. Ursache ist, daß die elektrischen Schaltkreise, aus denen ein Computer besteht, nur zwei Zustände annehmen können, nämlich Spannung da oder Spannung nicht da, Strom fließt oder fließt nicht, ein Transistor leitet oder sperrt. Ein paar Hunderttausend dieser Schaltkreise (Transistoren) bilden nun die CPU (Central Processing Unit, Zentraleinheit, praktisch das Herz des Computers), nochmals mehr als 8 Millionen davon (so Sie einen Mega-Amiga haben) sind der Speicher (das Gedächtnis) des Rechners.

Ein Programm ist nichts weiter als ein bestimmter Zustand dieses Speichers. Da es nun höchst unpraktisch ist, ein Programm in der Art zu beschreiben »Transistor 1 leitet, Transistor 2 auch, Transistor 3 sperrt, Transistor 4 leitet usw.«, kam man schnell auf eine Kurzschreibweise dieser Art: Der eine Zustand heißt 0, der andere 1. So kann man ein Programm doch schön kompakt schreiben, zum Beispiel als:

```
0101110011010101010 usw.
```

Das gefällt Ihnen nicht? Nun, das ist die Maschinensprache, mehr nicht !!!

Was Sie wohl schon erkannt haben: Dieses 0101011-Muster ist eine Zahl in dualer Schreibweise (da komme ich noch drauf zurück). Diese Zahlen kann man umrechnen in Dezimal- oder Hexadezimal-Zahlen, das spart etwas Papier, es bleibt aber Maschinensprache.

1.2 Höher: Assembler

Manche Leute behaupten nun, Assembler sei diese Maschinensprache. Gott sei Dank haben die unrecht, das wäre ja schrecklich. Die armen Kerlchen, die die ersten Computer so programmiert haben, tun mir heute noch leid.

Assembler ist die nächsthöhere Stufe und war einst der ganz große Fortschritt und viele Jahre lang auch die einzige Sprache überhaupt. Nun muß ich leider doch noch etwas ausholen.

Bits und Bytes

Wenn Sie bei Bit nicht mehr zuerst an Bier denken, sind Sie schon Programmierer, O.K... Ein Bit ist eine Speicherstelle, ein solcher Schaltkreis im Computer, der nur diese Zustände 0 oder 1 annehmen kann. Aus technischen Gründen hat man immer 8 Bit zusammengefaßt, diese 8 Bit nennt man ein Byte. Der Speicher eines Computers besteht aus tausenden oder Millionen von Bytes. Damit man nun jedes Byte ansprechen kann, sind sie durchnumeriert. Diese Hausnummern der Bytes nennt man Adressen. Mit den 8 Bit eines Bytes lassen sich in dualer Schreibweise die Zahlen 00000000 bis 11111111 darstellen, dezimal ist das 0 bis 255. In ein Byte (der Fachmann sagt, auf eine Adresse) kann ich nun eine solche Zahl hineinschreiben und sie wieder herauslesen. Die sogenannten Peripherie-Geräte wie der Bildschirm, die Tastatur oder ein Drucker sind nun mit einem Teil des Speichers (unseren Bytes) verbunden. Wenn ich auf die richtige Adresse eine Zahl schreibe, dann erzeugt sie eine Wirkung auf dem Bildschirm; wenn ich aus einer anderen Adresse etwas lese, dann kann das zum Beispiel eine Taste des Keyboards sein.

Bewegen ist alles

Folglich besteht ein Programm zum großen Teil daraus, Zahlen — man spricht auch von Daten — auf eine Adresse zu schreiben, von einer anderen zu lesen und ganz wesentlich, Daten von einer Adresse (zum Beispiel Tastatur) auf eine andere Adresse (zum Beispiel Bildschirm) zu kopieren.

Neben den Daten kennt so ein Computer auch Befehle, natürlich auch nur als 010101110, sprich als Zahlen.

Nehmen wir an, die Zahl 11111111 ist der Befehl »Kopiere«, und wir wollen Daten von der Adresse 0000011 (dezimal 3) auf die Adresse 00001001 (dezimal 9) kopieren, dann lautet dieses Programm in Maschinensprache

```
11111111
ØØØØØØ11
ØØØØ1ØØ1
```

In Assembler hingegen schreibt man dafür

```
MOVE 3,9
```

Move heißt bewegen, hier bewege, was im Byte mit der Adresse 3 steht, zum Byte mit der Adresse 9. Um gleich einen großen Denkfehler auszuschließen: Das Byte 3 bleibt unverändert, es wird nur in das Byte 9 kopiert. Sie haben recht, der Befehl müßte eigentlich COPY heißen, aber er heißt nun mal MOVE.

So, den Unterschied zwischen Assembler und Maschinensprache hätten wir; ist doch ein Fortschritt, oder?

Doch schon haben wir das nächste Problem. Der Begriff »Assembler« hat nämlich eine doppelte Bedeutung. Zum einen ist damit eine Programmiersprache gemeint, genauso wie zum Beispiel BASIC oder Pascal. Der Unterschied ist hauptsächlich, daß Assembler immer an eine bestimmte CPU gebunden ist. Es gibt zum Beispiel den Z80-Assembler, den 8088-Assembler und natürlich den 68000-Assembler, um den es hier geht. Die Sprache hat Befehle, wie alle anderen Sprachen auch, die Sie auch einfach so eintippen, wie üblich.

Der große Unterschied zu zum Beispiel BASIC ist dann nur, daß Sie danach nicht RUN eingeben können, sondern den Text erst assemblieren müssen. Genau das erledigt ein Programm, das dummerweise auch Assembler heißt. Dieses Programm übersetzt den Text in die Maschinensprache, also die 0101010-Folge, die die CPU letztendlich eh nur versteht.

1.3 Ganz oben: Hochsprachen

In einer Hochsprache, wie zum Beispiel Pascal, geben Sie auch nur Text ein; auch der muß übersetzt werden, nur heißt dann das Übersetzungsprogramm nicht Assembler, sondern Compiler. Das heißt, sowohl nach einem Assembler- als auch nach einem Compilerlauf entsteht ein Programm in Maschinensprache, das auf einem Computer ausgeführt werden kann. Über Größe und Schnelligkeit der Programme ist damit noch nichts gesagt.

Ganz anders sieht es bei einem Interpreter aus; der typischste Vertreter dieser Gattung ist wohl BASIC. Auch hier geben Sie das Programm als Text ein. Vielleicht wird es nach der Eingabe noch etwas aufbereitet und komprimiert, aber es bleibt Text, der nicht die geringste Ähnlichkeit mit Maschinensprache hat. Folglich kann der Computer ein BASIC-Programm auch nicht ausführen. Diese Aufgabe übernimmt der Interpreter. Er liest den BASIC-Text Zeichen für Zeichen und untersucht ihn auf BASIC-

Befehle. Findet er einen BASIC-Befehl, so ruft er eine Routine auf, die den Befehl ausführt. Die Routine befindet sich natürlich als ausführbares Maschinenprogramm im Speicher. Sie übernimmt es auch, zu einem BASIC-Befehl gehörige Daten (Parameter) im BASIC zu suchen. Selbstverständlich ist auch der Interpreter selbst ein Programm in Maschinensprache. Alle schnellen BASIC-Interpreter sind in Assembler geschrieben.

1.4 Assembler im Prinzip oder: warum so umständlich?

Ja, wenn denn nun der Compiler genauso Maschinen-Code erzeugt, wie ein Assembler, dann sollte ich mir die Sache doch noch einmal genau überlegen.

In Pascal zum Beispiel schreibe ich einfach nur

```
Write('Hallo')
```

und in Assembler tippe ich dafür (nur als Beispiel):

```
MOVE #'H',4711
MOVE #'a',4712
MOVE #'l',4713
MOVE #'l',4714
MOVE #'o',4715
```

Demnach ist ein Assembler-Programm die Auflösung von zum Beispiel Pascal-Befehlen wie WRITE in viele Einzelbefehle. Man kann es auch anders sagen: Pascal kennt eine bestimmte Menge von Befehlen, aus denen der Compiler die passende Folge von Assembler-Befehlen erzeugt.

Tatsächlich ist jedes Assembler-Programm (in der noch nicht übersetzten Textform) immer länger als sein Äquivalent in einer Hochsprache. Nur wenn Sie einmal nach dem Assemblieren bzw. dem Kompilieren jeweils die Bytes des Codes zählen, dann ist ein Assembler-Programm drastisch kürzer und schon deshalb auch schneller. Das liegt daran, daß kein Compiler einen so kompakten Code generieren kann, wie es ein Assembler-Programmierer tut. Letzterer weiß ja, was er will, er kann jede Befehlsfolge »maßschneidern«, ein Compiler hingegen muß Universallösungen einsetzen.

Ganz drastisch, im Tempo so bis zu Faktor 200, ist der Unterschied zu einem BASIC-Interpreter. Dieser übersetzt - wie schon geschildert - erst während der Laufzeit, und dann immer nur einen Befehl. Das heißt, wenn in einer Schleife ein Befehl 100mal wiederholt wird, dann wird er auch 100mal übersetzt. In einem Assembler-Programm hingegen ist der Befehl schon übersetzt.

1.5 Wann Assembler und wann besser nicht?

Nun mag ja manche Leute das Tempo nicht stören, sie haben Zeit, aber es gibt da noch einige Gründe.

Ein BASIC-Interpreter (oder ein Pascal-Compiler) kratzt eigentlich nur an der Oberfläche eines Riesenpotentials von Möglichkeiten, die in so einem Computer stecken. Will man mehr oder etwas anderes, dann muß man das der CPU nur sagen, allerdings in ihrer Sprache, und das ist nun mal Assembler.

Noch ein Grund: Man sollte eigentlich immer die Sprache verwenden, die das jeweilige Problem mit minimalem Aufwand löst. Oft genug, sogar meistens, ist das nicht Assembler. Ich möchte sogar fast behaupten, je besser man Assembler kann, desto weniger braucht man ihn. Ein Assembler-Programmierer weiß nämlich, was er mit welchen zum Beispiel BASIC-Befehlen der CPU an Arbeit zumutet und kommt so zwangsläufig zu besseren Programmen. Denn das muß ich nun leider auch noch erwähnen, Assembler setzt gute Kenntnisse der Funktion eines Rechners voraus.

Aber zum Trost: Diese Kenntnisse erwirbt man am besten, wenn man Assembler lernt.

Untersucht man nun ein Programm, das in einer Hochsprache geschrieben wurde oder geschrieben werden soll, dann stellt man fest, daß es nur an einigen Stellen (meistens nur an einer Stelle) das Tempoproblem gibt oder die passende Funktion fehlt. Dann sollte man auch nur diesen Teil in Assembler schreiben und ihn in die Hochsprache einbinden. Wie das geht, »kriegen wir später«.

Wie auch immer: Die Sprachen weit weg von der Maschine nennt man Hochsprachen, in Assembler sind wir »ganz unten«. Auch wenn Sie später nur noch in den höheren Regionen schweben, Sie wissen, mit einer soliden Grundausbildung schwebt es sich leichter, und man fällt nicht so leicht herunter.

1.6 Was man wofür an Software braucht

Die typische Arbeitsfolge einer Programmentwicklung in Assembler sind Texteingabe, Assemblieren, Linken (kommt gleich) und Testen.

Das sind Ihre Werkzeuge, und wie in jedem Handwerk kommt es darauf an, daß Sie mit den richtigen Werkzeugen arbeiten. Da gibt es nun leider eine große Auswahl, und die Prospekte der Hersteller versprechen alle viel. Ich möchte Ihnen hier einige Tips geben, die Sie bei der Auswahl beachten sollten und dann einige typische Erzeugnisse vorstellen. Vergessen Sie eines nie: Ein Assembler ist ein Profi-Werkzeug, das eine

gute Dokumentation und Support braucht. Natürlich bekommen Sie ein (versehentlich) kopiertes Spielprogramm auch per »Trial and Error« zum Laufen. Dies auf einen Assembler anzuwenden, dürfte nur etwas für Leute mit sehr guten Nerven und unendlich viel Zeit sein. Es kann nämlich durchaus sein, daß alle Programme in diesem Buch mit Ihrem speziellen Assembler nicht laufen, weil Ihr Assembler an einer Stelle einen Punkt verlangt, den meiner nicht braucht. Warum also knobeln, wenn alles im Handbuch steht.

1.6.1 Der Editor

Den Editor brauchen Sie für die Texteingabe und dessen Korrektur. Den Text nennt man Quelltext (Source-Text). Üblicherweise werden Editor und Assembler zusammen verkauft. Sie können aber auch ohne weiteres Ihr gewohntes Textverarbeitungsprogramm nehmen, wenn Sie sich auf reinen Text (keine Formatier- und Steuerzeichen) beschränken. Auch ED (gehört zum Amiga) ist dafür brauchbar.

Der Assembler meldet Ihnen Fehler mit einer Zeilennummer, der Text wird aber ohne Zeilennummern eingegeben. Demnach sollte der Editor ein »Go To-Zeile« können. Besser ist natürlich, wenn er selbst die Zeile mit dem Fehler anspringt. Da Sie Programme gegebenenfalls umgestalten und recht oft Textteile kopieren (und geringfügig ändern), sollte das Bewegen und Kopieren von Blöcken möglich sein.

1.6.2 Der Assembler

Ist der Text fertig (und auf der Diskette), starten Sie den Assembler, der dann mindestens wissen will, wie der File mit dem Quelltext (Source-File) heißt. Der Assembler erzeugt den Maschinen-Code (dieses 010101010), - auch Objekt-Code genannt - und legt diesen in dem Ziel-File (dem Objekt-File) auf der Diskette ab. Das kostet natürlich Zeit, und so erscheint es sinnvoll, auch »in memory« assemblieren zu können. Das heißt, der Assembler schreibt auf Wunsch den Code direkt in den Speicher, und man kann das Programm zu Testzwecken starten. Das Feature sollten Sie aber nicht überbewerten, denn das gleiche Ziel erreichen Sie auch mit einer RAM-Disk oder vom Zeitverhalten her gesehen auch mit einer Festplatte. In beiden Fällen muß natürlich der Assembler auch auf einer RAM-Disk bzw. einer Festplatte laufen. Darauf sollten Sie aber eh bestehen. Sonst sind noch folgende Eigenschaften wichtig:

»Include-Files«:

Der Assembler kann Textmodule einbinden. Das ist sehr wichtig, denn Amiga-Programme benötigen immer die sogenannten Libraries (Bibliotheken), die als Textmodule vorliegen. Außerdem haben Sie sich nach einer Weile selbst eine kleine Bibliothek von Routinen angelegt, die Sie in fast jedem Programm brauchen.

Makrofähig: Ausführlich werden Makros im Kapitel 6 behandelt. Hier nur soviel: Makros tragen stark zur Rationalisierung der Arbeit bei und helfen, Fehler zu vermeiden.

Fehlermeldungen: Schauen Sie im Handbuch nach. Je länger die Liste der Fehlermeldungen ist, desto besser werden Sie informiert.

Warnungen: Ein guter Assembler warnt Sie (berät Sie), wenn Sie nicht optimal programmiert haben. Auch hier gilt: je mehr »Warnings« desto besser.

1.6.3 Der Linker

Nun brauchen Sie den Linker, zu deutsch Binder. Der Binder hat zwei Aufgaben:

Zum einen können Sie ein Programm in Module aufteilen, die Sie getrennt assemblieren und testen können (bei sehr großen Programmen empfehlenswert). Diese Module müssen Sie dann mit dem Linker zu einem Programm zusammenbinden. Der zweite Grund liegt beim Amiga selbst. Jedes Programm hat einen kleinen Vorspann, Header genannt, in dem zum Beispiel steht, wie groß das Programm ist. Ohne diese Information kann der Amiga das Programm nicht laden und starten. Folglich muß der Linker zumindest diesen Header mit Ihrem Programm binden. Es gibt aber auch Assembler, die das schon tun. Das erspart den Linkerlauf, was man durchaus positiv sehen sollte. Besonders ALINK (der Standard-Linker) ist sehr langsam. Der Nachteil der fehlenden Modulisierung sollte dann aber durch »Include«-Fähigkeit und Makros ausgeglichen werden können.

1.6.4 Der Debugger

Starten Sie nun Ihr Programm, gibt es drei Möglichkeiten: Entweder es läuft oder es läuft nicht oder es läuft falsch. Um den Bug (Programmierer-Slang für Fehler) zu finden, bieten sich viele Lösungen an. Die einfachste (und meist erfolgreichste) Methode ist ein tiefer Blick auf den Quelltext kombiniert mit intensivem Nachdenken.

Wenn Sie aber wissen wollen, was das Programm an einer bestimmten Stelle tut oder welche Werte dann einige Variable haben, wird's schwierig. Möglich ist es, an diesen Stellen sozusagen ein »PRINT A,B« einzubauen, was aber in Assembler recht aufwendig ist, wie wir noch sehen werden (es gibt keinen Print-Befehl). Praktischer ist es dann, einen sogenannten Debugger (Entwanzer, Fehler sind Wanzen!) einzusetzen. Das ist ein Programm, mit dem Sie Ihr Programm in Einzelschritten ablaufen lassen und sich an jeder Stelle die Werte der Variablen ansehen können.

Versprechen Sie sich aber nicht zuviel von einem Debugger. So ein Programm ist gar nicht so einfach zu bedienen und wird Sie gerade in der Anfangsphase mehr verwirren, als daß es Ihnen hilft. Hinzu kommt, daß Beginner meistens Fehler begehen, die der Compiler schon findet. Nochmals, weil es so wichtig ist: Der Fehler steckt immer im Quelltext. Ein tiefer Blick darauf und intensives Nachdenken ist der beste Debugger! Wenn Sie einen Debugger erwerben, so müssen Sie auf zweierlei achten:

Zuerst sollte es ein symbolischer Debugger sein. Das bedeutet folgendes: Im Assembler-Programm arbeiten Sie niemals mit absoluten Adressen, sondern mit Labels (Marken), das sind dann die symbolischen Adressen. Der Assembler führt nun eine Tabelle, in der er zu den »Symbolen« die echten Zahlen notiert. Ein symbolischer Debugger greift nun einfach auf diese Symboltabelle des Assemblers zu. Daraus folgt nun die zweite Forderung, nämlich daß der Debugger das auch kann, sprich zum Assembler kompatibel ist.

1.7 Was man kaufen sollte (und was nicht)

Editor, Compiler, Linker und Debugger (soweit vorhanden) werden meistens im Paket angeboten (so sollte es sein). Häufig gehört dazu noch eine sogenannte Shell (eigene Benutzeroberfäche), die es Ihnen gestattet, zum Beispiel direkt vom Editor in die Shell zu wechseln, wo Sie dann den Linker aufrufen. Das ist, sofern man ohne RAM-Disk arbeitet, schneller als der Umweg über die Workbench bzw. das CLI. Ansonsten hat eine gute Shell den Vorteil, daß sie sozusagen eine für die Programmierung maßgeschneiderte Workbench ist.

1.8 Drei Assembler im Vergleich

Der folgende Abschnitt ist ein Testbericht, den ich schon in »Computer persönlich« veröffentlicht habe. Er setzt einige Dinge voraus, die erst später im Buch genauer erklärt werden. Lassen Sie sich dadurch nicht ins Grübeln bringen, sondern ignorieren Sie diese Passagen vorerst. Ich bin sicher, daß Sie nach diesem Abschnitt trotzdem genau wissen, wie die einzelnen Assembler Ihren Ansprüchen gerecht werden.

Erprobt wurden die Assembler von Metacomco, Kuma (K-SEKA) und HiSoft (DEVPAC Amiga). Der Metacomco-Assembler ist der Standard-Assembler für den Amiga. Das sollte jedoch kein Grund sein, genau diesen zu nehmen, denn die Konkurrenten müssen bekanntlich immer etwas bieten, was sie vom Standard vorteilhaft unterscheidet.

Bleiben wir also vorerst bei Metacomco, um so besser können wir dann auf die Unterschiede eingehen. Geliefert wird eine Diskette mit der Aufschrift »Macro Assembler for the Amiga, Version 11.00«. Das ist sicherlich nicht die 11. Amiga-Version. Der Hersteller liefert seit Jahren 68K-Assembler und paßt diese nur an verschiedene Rechner an.

Ansonsten beginnt der Frust beim Lesen des Handbuchs. Außer im Titel kommt nämlich das Wort »Amiga« im Manual praktisch nicht mehr vor, soll heißen, es wird mit keiner Silbe darauf eingegangen, wie man ein Assembler-Programm auf dem Amiga zum Laufen bringt.

Metacomco schwach dokumentiert

Es bleibt dem Leser überlassen, doch einmal auf der Diskette nachzusehen, wo er dann auch ein simples Beispiel findet, was aber die Fähigkeiten des Amiga (Grafik, Multitasking) völlig unberücksichtigt läßt. Der Umgang mit den beim Amiga eminent wichtigen Libraries wird nicht erwähnt, Dinge, wie der Unterschied zwischen CLI-Routinen, CLI-Tasks und Intuition-Tasks bleiben damit auch unerwähnt. Langer Rede kurzer Sinn: Wer nicht schon den 68000-Assembler im allgemeinen und den Amiga im besonderen beherrscht, hat keine Chance, anhand dieses Manuals das Programmieren des Amiga zu erlernen. Metacomco unterstellt offensichtlich, daß der Anwender über die komplette Amiga-Dokumentation verfügt (oder ein Buch wie dieses besitzt).

Der Erwerb des DOS-User-Manuals sowie des Developer's Manuals wird sogar ausdrücklich in der Einleitung empfohlen. Um so mehr verwundert es, daß sich die ersten 18 Seiten des Assembler-Manuals mit dem Editor ED befassen, der schon im DOS-Manual beschrieben ist (Metacomco liefert keinen eigenen Editor mit). Blieben die Seiten 19–50, auf denen der Assembler an sich vorgestellt wird. Hierbei handelt es sich um einen zwar konventionellen, aber grundsoliden 68K-Standard-Assembler, der punktgenau die Motorola-Spezifikation erfüllt. Die einzelnen Direktiven werden der Reihe nach gelistet. Auch hier wird unterstellt, daß der Leser weiß, was er damit anfangen kann; Beispiele fehlen nämlich. Die letzten acht Seiten schildern die Makro-Funktionen, die dem Kind auch den Namen gegeben haben. Dieser Makro-Teil ist lobenswert und für eine sinnvolle Amiga-Programmierung nahezu unabdingbar. Eine Unmenge von Makros ist auch Teil der mitgelieferten Include-Files, die sich übrigens präzise an die Listings im Kernel-Manual (Amiga-Dokumentation) halten. Makros können bis zu 36 Argumente (0..9, A..Z) übergeben werden, wobei Argument Nummer 0 immer für den Typ (B, W, L) reserviert ist. Makros können vorher definierte Makros aufrufen. Diese Schachtelung ist bis zu einer Tiefe von zehn erlaubt. Innerhalb eines Makros können Bedingungen geprüft und gegebenenfalls die Makro-Expansionen verlassen werden.

Bedingtes Assemblieren ist übrigens auch möglich.

SEKA: Weit weg vom Standard

Der SEKA-Assembler ist von der Idee her ganz phantastisch, nur die praktische Umsetzung der Idee ist unbefriedigend. SEKA ist ein Programm, das Editor, Assembler und Debugger vereinigt. Alle Teile sind permanent im RAM. Auch das Assemblieren erfolgt »in memory« und ist deshalb sagenhaft schnell. Leider wird nur ein Include-File mitgeliefert (DOS-Lib), und damit hätten wir die größte Schwachstelle des Systems angesprochen. Die Include-Files für typische Amiga-Programme (Grafik, Sound, Intuition zum Beispiel) fehlen nämlich. Wer nun auf die Idee kommt, sich diese Files »unauffällig« von Metacomco oder HiSoft zu besorgen, wird enttäuscht werden. Zum einen kennt nämlich der SEKA-Assembler keine Include-Anweisung. Die Files müssen in den Quelltext eingefügt werden, was zu sagenhaft langen Listings führen kann. Ich empfehle Ihnen als SEKA-Anwender, sich Ihre eigenen Include-Files anhand der LVO-Listen im Anhang dieses Buches aufzubauen.

Da sich diese Listen nach der Standard-Syntax richten, müssen Sie allerdings zwei Änderungen vornehmen: Der Unterstrich vor den Namen muß entfallen. Nach den Namen sind Doppelpunkte einzusetzen.

Zum zweiten beinhalten die Include-Files auch Makros. Diese kann SEKA zwar auch behandeln, nur leider verwendet er dafür eine andere Syntax. In diesem Punkt ist SEKA auch sonst konsequent, kräftige Abweichungen vom Standard sind die Regel. Im Kapitel 4 werden die Unterschiede anhand eines Beispielprogramms geschildert. Am Ende des Listings wird besonders deutlich, wie die Assembler-Direktiven SEKA vom Standard abweichen. Leider gibt es aber auch bei der 68K-Mnemonik Unterschiede. So erkennt SEKA zum Beispiel MOVEA nicht an, sondern besteht auf ein einfaches MOVE.

SEKA wird über Buchstaben-Kürzel gesteuert, die ein einfaches Umschalten zwischen Editor, Assembler und Debugger ermöglichen. Die Schwachstelle ist der Editor. Ursprünglich als Zeileneditor konzipiert, wurde er noch um einen einfachen Schirmeditor ergänzt. Letzterer ist aber recht spartanisch ausgestattet und recht langsam, so daß man doch sehr oft wieder in den Zeilenmodus zurückschalten muß.

Wiederum nur ein Tastendruck löst dann den Assembler aus, der mit einer so hohen Geschwindigkeit arbeitet, daß man bei mittelprächtigen Programmen eine Assemblierzeit praktisch nicht wahrnimmt. Ist das Programm fehlerfrei assembliert, sollte man es tunlichst auf der Disk sichern, denn der folgende Debug-Lauf könnte Probleme ergeben. Laut Handbuch reicht es zwar, auf das letzte Statement (hier RTS) einen Breakpoint zu setzen, doch soweit kommt das Programm im Debug-Modus oft gar nicht. Sobald das Programm nämlich auf einen Input wartet, hängt es sich schlicht auf. Hier hilft nur noch ein Neustart. Nun bin ich allerdings der Ansicht, daß Eingabe-Routinen wesentliche Bestandteile von Programmen sind. Was nützt mir also ein Debugger, der genau da nicht mitspielt?

Das Handbuch von SEKA ist mit 34 Seiten noch kleiner als das von Metacomco, trotzdem findet der Beginner dort mehr nützliche Informationen. Der Editor wird auf den Seiten 4-7 beschrieben, die Seiten 8-13 stellen den Assembler vor, 14-18 den Debugger. Nach zwei weiteren Seiten über File-I/O wird auf 21-23 der Linker vorgestellt.

Das Kapitel über den Linker habe ich zweimal gelesen, um dann festzustellen, daß es gar keinen Linker gibt, sondern nur den Assembler, der auch linken kann. Praktisch gibt es einen Bereich für den Objekt-Code und einen für den zu linkenden Code. Mit CL wird ersterer in den Link-Bereich kopiert oder mit RL ein Modul in den Link-Bereich eingelesen. Der Assembler erzeugt normalerweise ausführbaren Code, mit der L-Option hingegen generiert er linkbaren Code.

Man kann nun mit RL mehrere Module in den Link-Bereich einlesen (werden aneinandergehängt). Im Code-Bereich darf dann nichts sein oder der Quelltext des ersten Moduls, der dann vorab assembliert wird. Assembliert man nun ohne L-Option, wird das Ganze gebunden, vorausgesetzt man hat alle Module in absoluter Adressierung geschrieben. Da lobe ich mir doch einen Linker, den ich nur mit einer Liste aller zu bindenden Files versorgen muß, bzw. ihm sagen kann, aus welchen Bibliotheken er sich die fehlenden Module holen soll.

Idealer Linker nur bei HiSoft

Diese Eigenschaften besitzen sowohl der Linker von Metacomco als auch der von HiSoft. Metacomco setzt den Standard-Linker des Amiga ein. Der Linkerlauf muß bei Metacomco folgen, was sehr viel Zeit kostet. Bei HiSoft kann man wählen, ob der Assembler ausführbaren oder linkbaren Code erzeugen soll. Im Normalfall wird man auf letzteres verzichten. Will man doch Module einbinden, hat man allerdings einen Vorteil. HiSoft liefert einen Linker mit, der zu ALINK kompatibel ist, nur deutlich schneller. Wohl um den Fortschritt gegenüber ALINK anzudeuten, erhielt dieser Linker den sinnigen Namen BLINK (sprich B-Link und nicht Blink).

Damit wären wir auch schon beim dritten Produkt, nämlich dem DEVPAC von HiSoft angekommen. Um es gleich zu sagen, ich habe mir das Beste zum Schluß aufgehoben. Neben BLINK besteht das Paket aus dem Editor/Assembler GENAM, dem Debugger MONAM und dem kompletten Satz aller I-Files (voll mit Metacomco kompatibel) sowie einigen Demo-Programmen. Auch der wichtige Startup-Code liegt im Quelltext vor (ohne den Bug, den das Listing in der Amiga-Dokumentation hat). Zusätzlich gibt es noch ein Programm mit dem Namen GEMINST, das eine Voreinstellung bestimmter Parameter, wie zum Beispiel bevorzugte Größe des Textpuffers oder des Tabulator-Rasters erlaubt.

Beim Editor handelt es sich um einen Schirmeditor mit Pull-down-Menüs, die auf Assembler-Zwecke ausgelegt sind. Auffallend im Vergleich mit ED ist das sehr hohe Tempo, mit dem der Text gescrollt werden kann oder Such- und Ersetz-Funktionen

laufen. Der Code von GENAM ist mit 33 Kbyte sehr kompakt und wird deshalb auch erfreulich schnell von der Diskette geladen. Alles in allem ist das schon ein gutes Beispiel für die Vorteile der Assembler-Programmierung auf dem Amiga.

WordStar mit Maus

Über die Vor- und Nachteile mausbedienbarer Editoren ist schon viel geschrieben worden, hier erübrigt sich jede Diskussion. Der Cursor kann mittels der Pfeiltasten positioniert werden oder über Control-Codes, die mit WordStar kompatibel sind, oder auch mit der Maus. Letzteres ist sicher von Nutzen, wenn man eine Position anfahren will, die normalerweise diverse Einzelschritte erfordert.

Einen Fehler hat dieser Editor allerdings, den ED und auch K-SEKA nicht kennen: GENAM mag kein Deutsch, zumindest nicht auf einem Amiga 2000 mit deutscher Tastatur. Momentan habe ich in die »Startup-Sequence« ein »setmap usa0« eingebaut. Wenn ich dann blind tippe und mir dabei immer vorstelle, eine ASCII-Tastatur unter den Fingern zu haben, geht es. Trotzdem, ein Brief an HiSoft ist schon unterwegs.

Der Assembler wird vom Editor aus aufgerufen. Auch hierzu kann man den Befehl aus einem Pull-down-Menü ziehen oder ein Tastenkürzel (Amiga-A) tippen. Nach dem Assembler-Lauf ist man wieder im Editor. Liegt kein Fehler vor, kann man den Editor verlassen und das Programm, das dann schon ausführbar auf der Disk steht, aufrufen. Es gibt allerdings auch die Option, linkbaren Code zu erzeugen, der dann mit ALINK (schneller mit BLINK) noch zu binden ist. Hatte der Assembler Fehler festgestellt, werden diese im Klartext angezeigt. Nach deren Studium ist man mit einem Tastendruck wieder im Editor. Nun kann man die fehlerhaften Zeilen mit »GOTO Zeilennummer« anspringen. Noch schneller ist das Kürzel »Amiga-J« (Menü-Punkt »Jump to error«), was direkt auf die erste fehlerhafte Zeile führt. Zusätzlich gibt es noch die Option, ins Nichts zu assemblieren, womit eine schnelle Syntaxprüfung möglich ist.

Wie lange dauert es nun, bis ein ausführbares Programm auf der Diskette steht? Gewählt wurde das jeweilige Beispiel-Listing aus Kapitel 4, allerdings unter Nutzung der Include-Files (nur bei Metacomco und DEVPAC vorhanden).

Zuerst die Zeiten:

SEKA:	5 Sekunden
Metacomco:	95 (87) Sekunden
HiSoft:	11 (2) Sekunden

Metacomco verbraucht sehr viel Zeit mit dem Linken, Zeiten die bei den beiden anderen entfallen. Der reine Assembler-Lauf dauert bei SEKA nur Sekundenbruchteile, die Zeit ging für das Abspeichern auf die Diskette drauf. HiSoft schreibt direkt auf die Diskette. Es ist allerdings kein Problem, den Editor/Assembler plus diverse Libraries in

der RAM-Disk zu halten. DEVPAC belegt 33 Kbyte, MonAm 18 Kbyte, alle Libs zusammen (braucht man nie) verlangen rund 180 Kbyte, es bleibt also immer noch genug Platz für Source und Code. In dieser Betriebsart hat man praktisch die Vorteile von SEKA mit dem Komfort von DEVPAC kombiniert. Das ausführbare Programm entsteht dabei in knapp einer Sekunde. Ferner kann man bei DEVPAC auch ohne RAM-Disk auf Zeiten in der Größenordnung von 2 Sekunden kommen, wenn man die Include-Files durch EQU-Anweisungen im Text ersetzt, wie es die Listings in den Kapiteln 4 und 5 zeigen.

Mon Ami

Der Debugger von HiSoft heißt MonAm, ich übersetze das immer mit »mon ami«, weil es sich hier wirklich um einen freund(lichen) Debugger handelt. MonAm ist ein symbolischer Debugger, er arbeitet jedoch auch klaglos mit Code-Files ohne Symboltabelle. MonAm bietet zuerst alle Monitorfunktionen, kann disassemblieren, tracen, Breakpoints behandeln, kurz, alles, was gute Programme dieser Art so bieten, ist vorhanden. Auffallend ist nun, daß man mit MonAm praktisch nie abstürzen kann. Das liegt schlicht daran, daß er alle »Gurus« abfängt, sprich die Exception-Vektoren auf seinen eigenen Handler »verbiegt«. Konzeptbedingt kann MonAm ein Programm (Task) nur laufen (tracen) lassen, wenn der Task »schläft«. Der Versuch der Zuwiderhandlung läßt die Meldung »Task must be suspended« erscheinen.

MonAm setzt automatisch einen Breakpoint auf den ersten Befehl. Wenn man will, kann man von da aus Schritt für Schritt durch das Programm gehen. In einem Fenster erscheinen dann alle Registerwerte, im nächsten steht ein Speicherauszug in hex und ASCII (stellt man auf seinen Datenbereich). Darunter folgt ein Stück des Quelltextes mit einem Pfeil auf die aktuelle Zeile. Alles in allem heißt das, man kann zu jedem Befehl direkt ablesen, wie er auf Speichervariable und Register wirkt. Wer dann den Fehler nicht findet....

Bestes Manual hat HiSoft

Das Handbuch des DEVPAC ist mustergültig. Es beginnt mit präzisen Anweisungen zum Erstellen der Sicherungskopien. Wie man sein System für ein oder zwei Drives oder eine Harddisk einrichtet, wurde auch nicht vergessen. Dann folgt ein Blitzkurs, in dem anhand eines Musterprogramms auf der Diskette präzise erklärt wird, wie Editor, Assembler und Debugger zu bedienen sind. Nun weiß man, das alles läuft, und kann sich den einzelnen Kapiteln widmen, die alles weitere sehr präzise, gut verständlich und trotzdem kompakt beschreiben.

Im Anhang werden einige wichtige Grundlagen geschildert, speziell das für den Amiga wichtige Thema »Libraries«. Es folgt ein Schnellkurs »in Sachen CLI«, gefolgt von einer sehr detaillierten Anweisung, wie man sich eine neue CLI/DEVPAC-Disk einrichten muß, wenn man zum Beispiel von WB 1.1 auf WB 1.2 umsteigt.

Im Handbuch befindet sich noch ein kleines Buch, nämlich das »Programming Pocket Reference Guide« von Motorola, sprich die offizielle Dokumentation aller 68 Kbyte-Befehle.

Fazit: HiSoft, Sieger nach Punkten

Für mich eindeutig das beste Paket ist das DEVPAC von HiSoft. Begründung: schnell, komfortabel, kompatibel.

Wer Low-Level-Programmierung im wahrsten Sinn des Wortes treiben will, ist mit SEKA gut bedient. Gemeint ist damit: Wie einst auf dem C64, wo man die Grafik mit »Poke, Poke, Poke« programmierte, kann man das beim Amiga sehr schön in Assembler, nur daß das dann »Move, Move, Move« heißt. Durch dieses direkte Ansprechen der Amiga-Hardware lassen sich im Vergleich zu den durch zig Instanzen laufenden Intuition-Calls extreme Geschwindigkeiten erreichen. Da man solche Dinge interaktiv programmieren muß (Operand ändern, Wirkung beurteilen), ist dann so etwas wie SEKA ideal, der einen die Assembler-Zeiten vergessen läßt.

Sobald man allerdings komplexere Aufgaben zu lösen hat, die in Richtung Strukturierung und Rationalisierung der Arbeit laufen, sollte man zu einem klassischen Assembler greifen. Da beklage ich bei Metacomco das zwar solide, doch etwas überholte und langsame Konzept. So bleibt also für mich DEVPAC als Alternative übrig.

Im Buch schildere ich für die ersten Listings die Unterschiede, die Sie beachten müssen, wenn Sie mit anderen Assemblern arbeiten. Alle weiteren Listings sind dann mit dem HiSoft-Assembler geschrieben worden.

Kapitel 2

Aufbau eines Computers

Register

Stapel

2.1 Das Computermodell

In diesem Kapitel müssen wir uns etwas mit dem Aufbau und der Funktion eines Computers beschäftigen. Wie das System elektrisch funktioniert, ist dabei aber völlig uninteressant. Für die Programmierung reicht immer ein sogenanntes Modell. Sie müssen wissen, was CPU, RAM, ROM und Bus bedeuten und leisten. Innerhalb der CPU interessiert dann ganz besonders das Registermodell.

Sie sind in der Situation eines angehenden Autofahrers. Ich erkläre Ihnen jetzt, welchen Sinn Lenkrad, Kupplung, Gaspedal und Bremse haben.

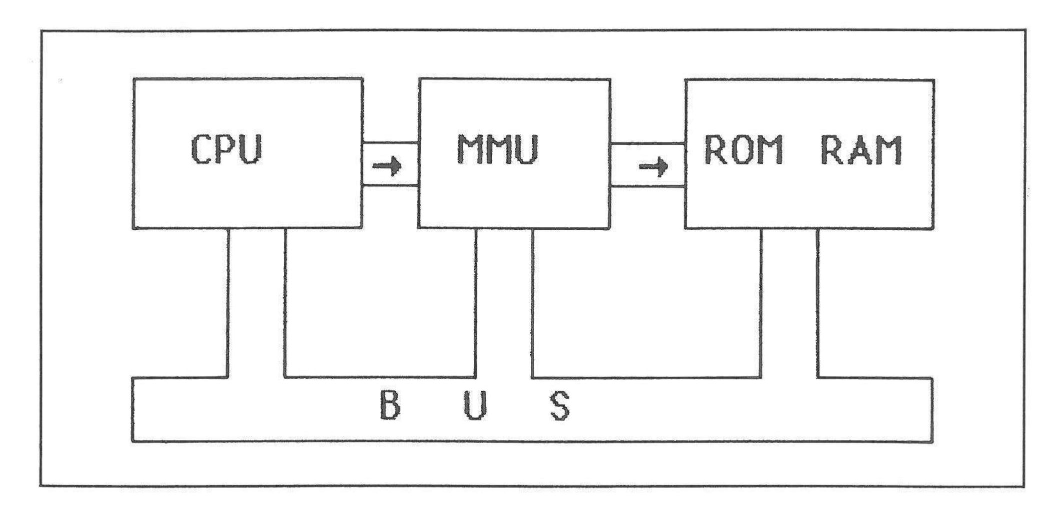


Bild 2.1: Modell eines Computers

Bild 2.1 zeigt das Modell eines Computers ziemlich vereinfacht. Der wichtigste Baustein ist die CPU (Central Processing Unit) oder der Prozessor, also der 68000. Er ist für die gesamte Ablaufsteuerung verantwortlich, er kann rechnen, entscheiden und vergleichen. Von allein tut er allerdings nicht viel, er braucht dafür ein Programm.

Der zweite große Baustein ist der Speicher, unterteilt in die Teile RAM und ROM. RAM heißt historisch Random Access Memory, also Speicher mit wahlfreiem Zugriff (man kann direkt auf jede Speicherstelle zugreifen, und nicht wie zum Beispiel beim Bandspeicher nur seriell), nur diese Eigenschaft hat ein ROM auch. Der große Unterschied: Beim RAM ist Lesen und Schreiben möglich, beim ROM (Read Only Memory) nur Lesen. Noch ein Unterschied: Der RAM-Inhalt ist verloren, wenn Sie den Computer ausschalten, der ROM-Inhalt ist permanent vorhanden.

Unsere Programme werden immer im RAM liegen, wir werden aber den ROM (so Sie einen Amiga mit Kickstart-ROM haben) kräftig nutzen. Um eine Speicherstelle im RAM oder ROM ansprechen zu können, muß die CPU diese Speicherstelle adressieren. Diese Adressen laufen über den Adreßbus.

Der Bus ist nichts weiter als eine Menge von Leitungen, über die alle Teilnehmer parallel geschaltet sind. Der Ausdruck Bus kommt daher, weil bildlich gesehen eine Information (zum Beispiel Adresse) an der Haltestelle CPU einsteigt, auf dem Bus fährt und dann an der Haltestelle RAM (oder ROM) aussteigt. Sinngemäß laufen die Daten (was in die adressierten Bytes hinein soll/aus ihnen gelesen wird) über den Datenbus.

Der Speicher selbst besteht aus vielen gleichartigen Chips. Diese haben alle den gleichen kleinen Adreßbereich, aber auch einen Eingang (Chip Select), über den man einen Chip anwählen kann. Deshalb muß eine logische Adresse in eine physikalische Adresse umgesetzt werden. Dies macht ein sogenannter Adreßdekoder oder - wie beim Amiga - eine leistungsfähigere Version davon, die sogenannter MMU (Memory Management Unit).

Für uns ist wichtig zu wissen, daß ein Zugriff auf geschützte Bereiche oder illegale Adressen mit einem »Bus-Error« (Guru-Meldung) bestraft wird.

2.2 Fetch & Execute

Generell läuft ein Programm in einem Computer nach der Methode »Fetch and Execute«, wie die Amerikaner so schön prägnant sagen. Auf deutsch heißt das »Holen und Ausführen«. Die CPU holt sich aus dem Speicher einen Befehl und führt ihn aus. Danach holt sie sich automatisch den nächsten Befehl und führt diesen aus, usw., usw. Natürlich muß im Speicher etwas stehen, das die CPU holen und ausführen kann, und das nennt man dann Programm.

2.3 Programme sind nur Bytefolgen

Ein Programm ist nichts weiter als eine Folge von Bytes, die irgendwo im RAM oder ROM steht. Natürlich kann die CPU nicht wissen, wo das Programm im Speicher steht. Deshalb wird sie beim Start (Reset) per Hardware-Vorgabe sozusagen mit der Nase auf eine Anfangsposition gestoßen. Ab diesem Augenblick holt sich die CPU immer ein Wort (das sind 2 Byte nebeneinander, also 16 Bit) aus dem Speicher und dekodiert dieses Wort. Dabei kommt dann (hoffentlich) ein Befehl für die CPU heraus. Diesen

Befehl arbeitet sie ab und holt dann das nächste Wort. Zu einem Befehl können Daten gehören. Beim Addierbefehl zum Beispiel muß die CPU wissen, was addiert werden soll. Wieviel Datenwörter zu einem Befehl gehören, ist auch im ersten Wort (dem Befehlswort) kodiert.

Der gesamte Speicher ist Byte für Byte von Null bis zum Ende durchnumeriert; diese Nummern der Speicherplätze nennt man Adressen. Die CPU arbeitet immer nur mit diesen Adressen und führt dazu intern einen Zähler, der immer auf die aktuelle Adresse zeigt, bei der sie gerade ist. Diesen Zähler nennt man »Program Counter«, kurz PC.

Hier ein Beispiel:

Adresse (PC)	Befehl	Daten
1000	Lösche	Wort
1004	Addiere	Operand 1, Operand 2
1010	Return	
1012		

Das »Listing« zeigt schematisch ein Programm, das bei Adresse 1000 beginnt. Um das Programm zu starten, muß man nur den PC auf 1000 setzen, und schon läuft es. Befehl 1 belegt die Adressen 1000 und 1001. Er hat in diesem Beispiel ein Datenwort auf Adresse 1002 und 1003. Die CPU arbeitet diesen Befehl ab und stellt dann den PC auf Adresse 1004. Zu Befehl 2 (auf 1004 und 1005) gehören zwei Datenwörter (1006-1009), folglich muß Befehl 3 bei Adresse 1010 starten.

Der 68000 kennt Befehle ohne Daten, die sind dann ein Wort lang, aber auch solche mit bis zu vier Datenwörter. Das heißt, beim 68000 kann ein einziger Befehl mit seinen Daten bis zu 10 Byte (5 Wörter) belegen. Wie Sie aus diesem Schema ersehen können, muß jeder Befehl auf einer geraden Adresse (Wortgrenze) beginnen, andernfalls passiert Übles.

Nun fragen Sie, wie das kommt. Ganz einfach: Sie können (und müssen) den PC verändern. Wenn nämlich ein Programm nicht nur einfach Befehl für Befehl abläuft, Sie also zum Beispiel ein GOTO benötigen, dann heißt das in Assembler zuerst einmal »GOTO Adresse«. Praktisch heißt das aber für die CPU »Setze PC = Adresse«. Wenn Sie da eine ungerade Adresse angeben, stürzt leider Ihr Programm ab.

In der Praxis tritt dieser Fehler auf, wenn Sie im Programm Daten definieren. Wenn Sie zum Beispiel den Text »Franz Meier« drucken wollen, müssen Sie irgendwo im Speicher eine Bytefolge mit den ASCII-Codes dieser Zeichen laden. Folgt dann der Text »8000 München«, und Sie wollen diesen Text einmal allein drucken, dann sollten

Sie wissen, wie lang »Franz Meier« ist. Um diese Abzählerei zu ersparen, haben gute Assembler einen Befehl (EVEN oder CNOP), der Texte (oder Daten allgemein) auf eine gerade Adresse justiert. Ist die Adresse sowieso gerade, passiert nichts. Ein »Even« zu viel schadet also nichts, eines zu wenig dagegen sehr.

So informiert werden Sie also nie wieder vor einem Text den EVEN-Befehl (oder Gleichartiges) vergessen, wie das die anderen Anfänger tun, oder?

2.4 User- und Supervisor-Modus

Der 68000 kennt zwei Betriebsarten mit den Bezeichnungen User-Modus und Supervisor-Modus, frei übersetzt: Anwender und Boß.

Im Supervisor-Modus laufen Kernroutinen des Betriebssystems. Unsere Programme (und Anwenderprogramme überhaupt) werden im allgemeinen im User-Modus ablaufen. Für Sie ist wichtig zu wissen, daß es einige 68000-Befehle gibt, die nur im Supervisor-Modus erlaubt sind. Diese in den Manuals als privilegiert bezeichneten Befehle dürfen Sie nicht anwenden, ohne vorher in den Supervisor-Modus umgeschaltet zu haben. Andernfalls wird Ihr Programm mit einer Guru-Meldung aussteigen. Das Betriebssystem des Amiga, speziell der Multitasking-Kern, reagiert sehr empfindlich auf Eingriffe von außen. Sie sollten deshalb den Supervisor-Modus meiden. Ein Manko ist das praktisch nicht, denn für spezielle Eigenschaften des 68000, die nur im Supervisor-Modus zugänglich sind, stellt Ihnen das Betriebssystem Routinen zur Verfügung, die Sie problemloser benutzen können.

2.5 Das hexadezimale Zahlensystem

Das hexadezimale Zahlensystem ist in Assembler üblich (und sehr vorteilhaft), machen Sie sich bitte gegebenenfalls mit diesem Zahlensystem vertraut. Hier ein Schnellkurs:

Die Basis ist nicht 10, wie im 10er-System, sondern 16. Für die nun fehlenden »Ziffern« von 10 bis 15 schreibt man A bis F. In dezimal sagt man für die Zahl 345 auch 5 Einer plus 4 Zehner plus 3 Hunderter. In hex ist die Basis 16.

Die Folge wäre also nicht 1, 10, 100, 1000 sondern 1, 16, 256 , 4096.

Sie wissen, F hat den Wert 15. Demnach ist
FFFF = 15 * 4096 + 15 * 256 +15 * 16 + 15 * 1 = 65535.

2.6 Ein BASIC-Programm zum Üben der Hexerei

Bild 2.2 bringt ein kleines Programm in Amiga-BASIC zum Üben.

```
While 1
     Input "Eine Zahl n ($n wenn hex) ";A$
     If Left$(A$,1)<>"$" Then
         Print Hex$(VAL(A$))
     Else
        A$=Right$(A$,Len(A$)-1)
        L=Len(A$)
        X%=0
        For I=L To 1 Step -1
           X%=X%+Val("&h"+Mid$(A$,I,1))*16^(L-I)
        Next I
        Print X%
      Endif
Wend
```

Bild 2.2: Hex-Dezi-Konvertierung in BASIC

Wenn Sie eine Dezimalzahl eingeben, dann gibt sie das Programm in hex aus. Geben Sie eine Hex-Zahl ein (erkenntlich am $ als erstes Zeichen), erhalten Sie deren Wert in dezimal.

2.7 Das duale Zahlensystem

Sozusagen noch eine Stufe tiefer (noch näher am Computer) ist das duale Zahlensystem. Hier ist die Basis 2, womit in diesem System nur die Ziffern 0 und 1 erlaubt sind. Am einfachsten kann man eine Dualzahl in dezimal umrechnen, indem man sich die Wertigkeit darüber schreibt. Hier ein Beispiel:

Dezimale Wertigkeit:	32	16	8	4	2	1
Dualzahl:	1	0	1	1	0	1

Das Ergebnis wäre dann 32+8+4+1 = 45

Die Verbindung zum hexadezimalen Zahlensystem ist recht einfach zu erledigen. Nehmen wir an, wir hätten diese Dualzahl:

```
1010 0101
```

Sie sehen schon, ich habe sie in Vierergruppen geteilt. Lege ich wieder die Wertigkeit darüber, sieht das so aus:

8	4	2	1		8	4	2	1
1	0	1	0		0	1	0	1

Das ergibt (von links) dezimal 10 und 5. In hex schreibt man für 10 aber A, also hieße die Zahl in hex A5. Auch hier wieder mit Bild 2.3 ein Amiga-BASIC-Programm zum Üben von Dualzahlen, auch Binärzahlen genannt:

```
WHILE 1
  INPUT "Eine Zahl n (%n wenn binär) "; a$
  IF LEFT$(a$,1) <> "%" THEN
     x%=VAL(a$)
     FOR i=15 TO 0 STEP -1
       PRINT SGN(x% AND 2^i);
     NEXT  : PRINT
  ELSE
    a$=RIGHT$(a$,LEN(a$)-1)
    l=LEN(a$): x%=0
    FOR i=l TO 1 STEP -1
      x%=x%+VAL("&h"+MID$(a$,i,1))*2^(l-i)
    NEXT
    PRINT x%
  END IF
WEND
```

Bild 2.3: Umrechnung von binär in dezimal und zurück

Wenn Sie eine Dezimalzahl eingeben, dann gibt sie das Programm in binär aus. Geben Sie eine Binärzahl ein (erkenntlich am % als erstes Zeichen), erhalten Sie deren Wert in dezimal. Das Zeichen Prozent (%) ist in Assembler der Präfix für Dualzahlen.

2.8 Stack: Funktion und Aufgaben

Das kürzeste Programm, das Sie für den Amiga schreiben können, heißt in Assembler:

```
CLR   -(SP)
```

und schon haben Sie den Stack benutzt. SP (oder A7, was dasselbe ist) werden Sie am häufigsten in jedem Programm finden; ein Grund, uns auch das Ding genauer anzusehen. Der Stack ist ein Speicher (ein Stück RAM) mit besonderen Eigenschaften. Man nennt ihn auch LIFO für »Last In, First Out« oder Stapelspeicher. Sie packen Daten auf den Stack, indem Sie etwas auf den Stapel tun. Sie können immer nur von oben (vom »Top of Stack«) etwas wegnehmen. Das heißt, wenn Sie die Daten A, B und C in die Reihenfolge auf den Stack packen, können Sie sie nur in der Folge C, B, A zurücklesen. Der Trick ist nun, daß die CPU tatsächlich niemals die Daten vom Stack nimmt, sondern nur die Daten woandershin kopiert. Gesteuert wird dieses durch den sogenannten Stapelzeiger, neudeutsch Stackpointer oder kurz SP. Noch eine Vorbemerkung: Der Stack wächst von oben (den hohen Adressen) nach unten (zu den niedrigen Adressen hin). Die Anweisung »Packe A auf den Stack« bewirkt zwei Schritte:

1. Erniedrige SP
2. Kopiere A in den Speicherbereich, auf den SP nun zeigt

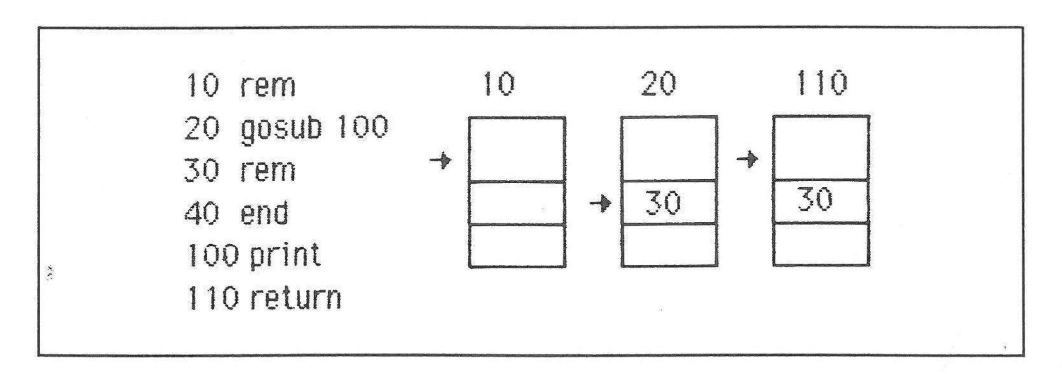


Bild 2.4: Sinn des Stacks am Beispiel BASIC

Die Umkehr, nämlich hole A vom Stack, hat zur Folge:

1. Kopiere Daten, auf die SP zeigt, nach A
2. Erhöhe SP

Was das unter anderem für einen Sinn hat, soll Bild 2.4 zeigen. Es geht um Unterprogramme, hier am Beispiel von BASIC. Sie können sich aber auch die Zeilen-

nummern als Adressen vorstellen. Rechts ist immer ein Stück des Stacks, daneben der Stackpointer gezeichnet. Der Befehl »GOSUB 100« bewirkt dreierlei:

1. Erniedrige SP
2. Packe die nächste Zeilennummer (hier 30) auf den Stack (auf die Speicherstelle, auf die jetzt SP zeigt)
3. Springe zur Zeile 100

Das Return in Zeile 110 hat zur Folge:
1. Hole Zeilennummer, auf die SP zeigt
2. Erhöhe SP
3. Springe zur Zeile 30

Nun fragen Sie vielleicht, warum SP vom GOSUB erniedrigt und vom RETURN erhöht wird? Nun, schauen Sie auf Bild 2.5. Hier ruft das Unterprogramm ein weiteres Unterprogramm auf. Jetzt stehen nach Zeile 110 zwei Zeilennummern (genau Return-Adressen) auf dem Stack. Das Return von Zeile 210 stellt den SP auf Zeile 30 und springt dann zu 120, das Return in Zeile 120 stellt den SP wieder zurück und springt dann zu Zeile 30. Der SP steht wieder auf seinem Ausgangswert, »we are home «.

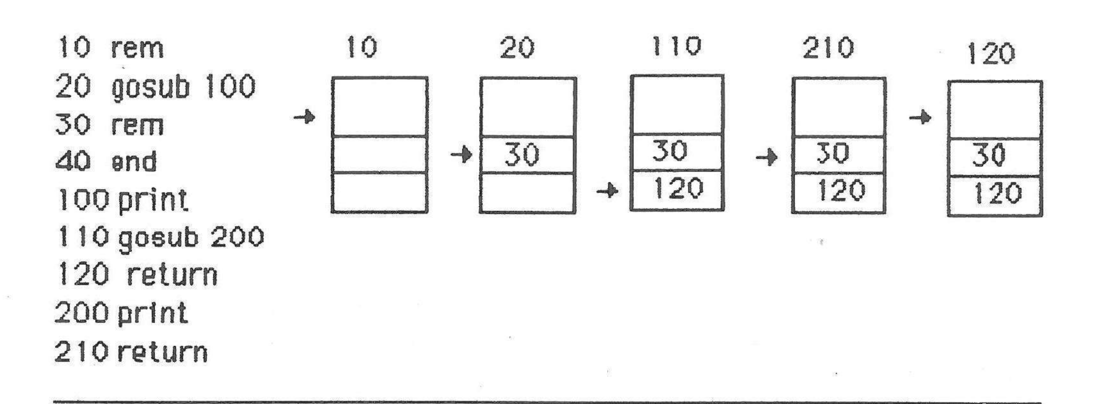


Bild 2.5: Stack im Fall »Unterprogramm ruft Unterprogramm«

Langer Rede kurzer Sinn, mit dem Stackmechanismus können Unterprogramme beliebig tief geschachtelt werden. Jedes Return erhöht den SP wieder und so hangelt man sich dann zurück. Aber Achtung, was passiert hier?

```
10 GOSUB 20
20 GOSUB 10
```

Da jedes GOSUB den SP erniedrigt, aber das Gegenstück, nämlich das RETURN fehlt, wächst der Stack nach unten. Er wird dann recht bald in Ihren Programm-Code laufen und den mit Return-Adressen überschreiben. Ergebnis: Totaler Crash, auch in BASIC. Probieren Sie es einmal. Die zweite Anwendung für den Stack ist die Parameterübergabe an Unterprogramme. Prinzipiell läuft das so: Es gibt in Assembler (nicht in BASIC) die Befehle »Packe Daten auf den Stack« und »Hole Daten vom Stack«. (Sie wissen, jeder Befehl impliziert ein Verändern des Stackpointers.) Da kann ich dann sinngemäß schreiben:

```
10 A auf den Stack
20 B auf den Stack
30 GOSUB 100 (Return-Adresse auf Stack)
```

und dann im Unterprogramm:

```
100 Hole Return-Adresse vom Stack (und merke sie)
110 Hole B vom Stack

120 Hole A vom Stack
130 Rechne mit A und B
140 Springe zur Return-Adresse
```

Was aber, wenn Ihr Unterprogramm mit Return enden soll? Dann schreibt man:
1. Return-Adresse auf den Stack
2. Daten auf den Stack
3. GOTO Unterprogramm

Im Unterprogramm:
1. Daten vom Stack
2. Mit Daten arbeiten
3. RETURN

Wie schon gesagt: Es gibt in Assembler keinen PRINT-Befehl, sondern nur die Möglichkeit, Bytes in einen Speicherbereich zu schreiben, der (vom Video-Kontroller) auf dem Bildschirm abgebildet wird. Überhaupt heißt Assembler-Programmierung primär, Daten von einer Adresse auf eine andere Adresse zu bewegen. Auch die Peripherie-Geräte (Tastatur, Floppy usw.) liegen beim Amiga innerhalb des Adreßbereichs (man nennt das »memory mapped«). Die Geräte werden angesprochen, indem man bestimmte Daten in diese Adressen schreibt oder von ihnen liest.

Praktisch werden wir zwar die Hardware kaum so ansprechen, sondern die Parameter in Datenstrukturen eintragen und dann System-Routinen aufrufen, aber auch diese Datenstrukturen müssen wir adressieren.

Sie sehen also schon, die Adressierung als solche ist ganz wesentlich. Man kann eine Adresse auf sehr viele unterschiedliche Arten ansprechen, ein Beispiel hatten wir schon mit dem SP. Ich kann da sagen, stelle den SP auf die Adresse 4711. Ich kann aber auch sagen, hole Daten von der Adresse, auf die SP gerade zeigt (ohne zu wissen, wohin er zeigt).

Das waren schon zwei Adressierungsarten. Insgesamt kennt der 68000 aber 12, und mit diesen 12 Adressierungsarten werden wir uns im nächsten Kapitel beschäftigen. Sie sind sozusagen der Schlüssel zum 68000.

Kapitel 3

Adressen, Daten und Befehle

Register, Adressierungsarten

Datentypen

Struktur der Befehle des 68000

Bisher hatten wir gelernt, daß Daten im RAM oder ROM stehen. Daneben gibt es aber einen ganz speziellen RAM, der ein Teil der CPU ist. Dieser Speicherbereich besteht aus Gruppen von je 32 Bit, und jede dieser Gruppen nennt man Register.

3.1 Tempo durch Register

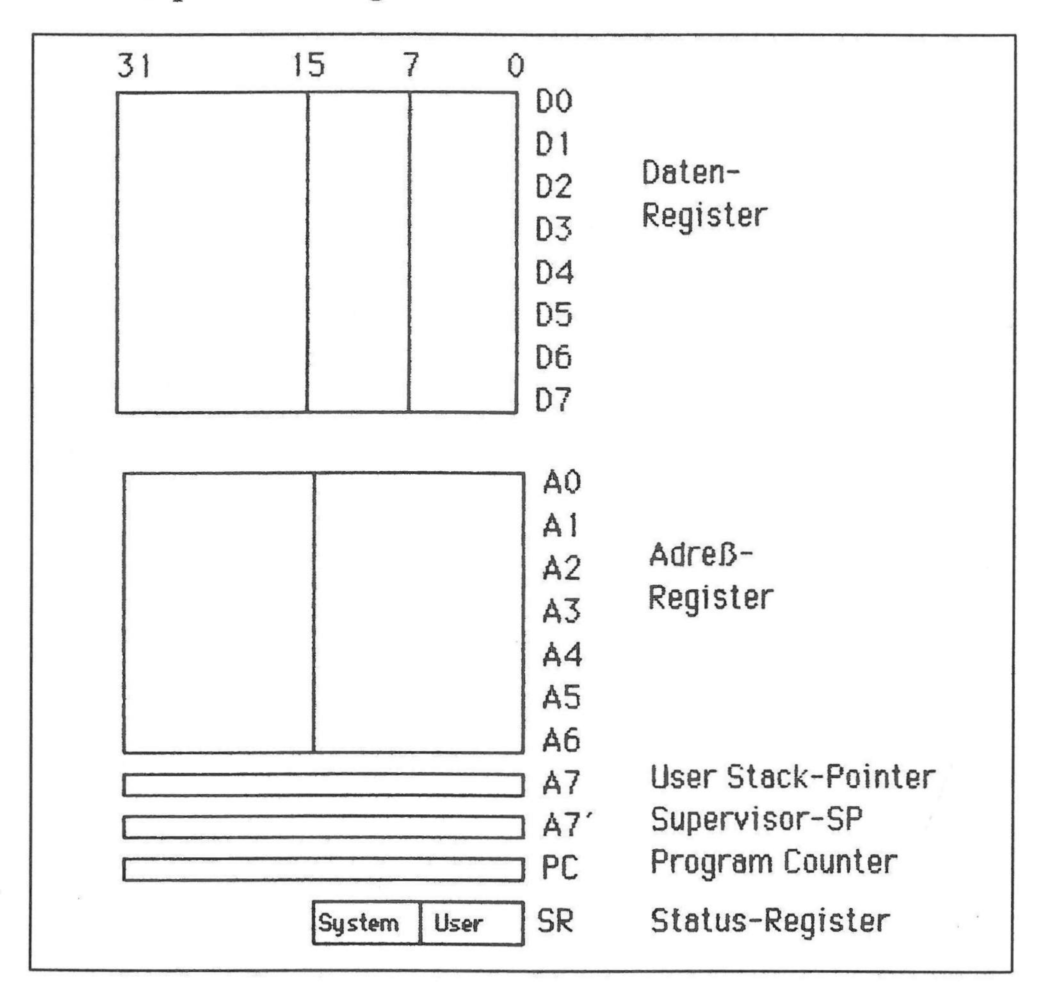


Bild 3.1: Das Registermodell des 68000

Die Register werden nicht über Adressen sondern über Namen angesprochen. Der Vorteil von Registern im Vergleich zum übrigen Speicher ist, daß sich die Register auf

demselben Chip wie die CPU befinden, und die CPU den Zugriff auf diese Register durch spezielle Befehle unterstützt.

Natürlich entfällt auch der Umweg über die MMU und den Bus. Damit sind Registeroperationen wesentlich schneller als Zugriffe auf den Hauptspeicher und bieten (wegen der speziellen Befehle) einiges mehr an Komfort.

3.2 Das Registermodell des 68000

So gesehen ist eine CPU mit vielen Registern besser, als eine solche mit wenigen. Der 68000 hat viele Register, nämlich:

acht Datenregister, sieben Adreßregister, zwei Stackpointer, einen Programmzähler (PC) sowie ein Statusregister. Bild 3.1 zeigt den kompletten Registersatz des 68000. Wie Sie sehen, sind die Register D0-D7 und A0-A7 plus PC je 32 Bit breit. Im Bild sind die Bits von 0 bis 31 numeriert. Das sind 4 Byte.

3.3 Datentypen

In einem Byte zählt man die Bits von 0 (niederwertiges Bit) bis 7 (höchstwertiges Bit). Zwei Byte (16 Bit) nennt man Wort. Dessen Bits zählen von 0 bis 15. Zwei Worte (32 Bit) sind ein Langwort. Man spricht auch von den Datentypen Bit, Byte, Wort und Langwort. Die größten darstellbaren Zahlen in Abhängigkeit vom Typ zeigt Bild 3.2.

Bit = 2^1-1 = 1
Byte = 2^8-1 = 255
Wort = 2^16-1 = 65535
Langwort = 2^32-1 = 4,294,967,299

Bild 3.2: Die Datentypen und damit darstellbare Werte

Nun verstehen Sie auch die senkrechten Trennlinien in Bild 3.1. In den Datenregistern kann man Bytes, Worte und Langworte ablegen. Bei den Adreßregistern ist der Typ Byte nicht möglich. Die Stackpointer (A7) sind immer »long«.

Außerdem gibt es noch den Type BCD (Binary Coded Decimal). In diesem Fall wird ein Byte in 2 Halbbyte (nennt man Nibble) geteilt. Mit den 4 Bit eines Nibbles kann man nun die Zahlen 0–15 darstellen. Auf 10 bis 15 wird dabei aber verzichtet, gültig sind in der BCD-Darstellung nur die Werte von 0 bis 9. Damit kann man in einem »BCD-Byte« immer zwei Zehnerstellen darstellen. Ein Wort reicht also für eine 4-stellige Dezimalzahl. Braucht man mehr Stellen, muß man die entsprechende Anzahl Bytes sozusagen nebeneinanderlegen. Nun gibt es eine Unmenge kluger Algorithmen zum Thema BCD-Rechnerei. Für die armen Leute ohne 68000 sind die sehr wichtig. Wir können darauf verzichten, weil im 68000 die passenden Befehle dafür schon eingebaut sind.

Typangabe ist die erste Bürgerpflicht

Solange Sie nur mit Registern arbeiten, spielt die Größe (fast) keine Rolle, wenn Sie aber ein mit allen 32 Bit gefülltes Register in den RAM kopieren, belegt es da 4 Byte. Das ist ziemlich unpraktisch, weil Sie sehr oft mit Bytes oder Worten auskommen könnten, also auch mit weniger Speicher. Deshalb gibt es beim 68000 die Möglichkeit – genauer: die Pflicht – bei jeder Operation, die Daten bewegt, anzugeben, welcher Typ dabei gilt.

Ein Beispiel: Der Datentransfer geschieht mit dem Befehl MOVE und zwar mit der Syntax »MOVE Quelle,Ziel«. Tatsächlich bewegt der Befehl MOVE die Daten nicht, sondern kopiert sie, er kopiert von der Quelle auf das Ziel.

Um die Daten im Register D3 auf die Adresse 4711 zu kopieren, schreibt man:

```
MOVE.B D3,4711   oder
MOVE.W D3,4711   oder
MOVE.L D3,47711
```

Im Fall .B wird ein Byte kopiert, also die Bits 0–7 von D3, im Fall .W ist es ein Wort (Bit 0–15) bzw. im Fall .L das ganze Register. Im Hauptspeicher (hier ab Adresse 4711) werden dann entsprechend 1, 2 oder 4 Byte belegt. In welcher Reihenfolge dabei die verschiedenen Datentypen im RAM stehen, sollten Sie wissen, nämlich so, wie man sich das denkt. Steht beispielsweise in D0 das Wort $AABB und wird D0 mit einem MOVE.W-Befehl auf die Adresse 1000 kopiert, so steht $AA in 1000 und $BB in 1001. Nur den Kollegen aus der 8-Bit-Ecke und denen, die vom IBM-PC kommen, sei noch einmal deutlich gesagt: Der 68000 speichert Daten in der richtigen Reihenfolge und nicht wie die »8-Bitter/8088er« die Bytes eines Wortes vertauschen!

Wegen dieser freien Auswahl des Datentyps haben Sie leider auch die Pflicht, ihn bei den meisten Befehlen anzugeben. Fehlt der Typ, nehmen die meisten Assembler den Typ Wort an. Um auf die Register zurückzukommen: Der Hauptunterschied zwischen

Daten- und Adreßregistern ist, daß bei letzteren die Typen Bit und Byte nicht erlaubt sind.

Ansonsten können Sie durchaus auch Daten in Adreßregistern speichern und Adressen in Datenregistern. Letzteres werden wir beim Amiga häufig antreffen, da viele Routinen auch Adressen in Datenregistern erwarten.

Das Statusregister hat einen ganz besonderen Zweck. Damit wird in Assembler IF-THEN realisiert; ich komme noch (sehr ausführlich) darauf zurück.

3.4 Befehle

Wieviel Assemblerbefehle es gibt, ist gar nicht so einfach zu sagen. Das liegt daran, daß ein Befehl je nach Adressierungsart (und weiteren Varianten) ganz unterschiedliche Wirkung zeigt. Beginnen wir mit dem Befehlsaufbau. Ein Befehl kann haben: keinen Operanden oder einen oder zwei. Die Operanden können im Befehlswort selbst enthalten sein oder belegen bis zu vier weitere Worte, die dem Befehlswort unmittelbar folgen. Darüber brauchen Sie sich aber vorerst wenig Sorgen zu machen. Im Quelltext schreiben Sie den Befehl und die Operanden einfach hin, wieviel Worte das dann werden, ist Sache des Assemblers.

Ein Beispiel für einen Befehl mit keinem Operanden ist RTS (Return from Subroutine), was dem RETURN in BASIC entspricht. Einen Operanden hätte der Befehl »CLR D0«. Das heißt Clear (Lösche (fülle mit Nullbits)) den Operanden D0 (das Register D0). Ein Beispiel für einen Befehl mit zwei Operanden:

```
MOVE.L  A3,A4
```

Damit wird das Langwort im Register A3 nach A4 kopiert.

3.5 Sinn und Zweck der Adressierungsarten

Eine CPU ist um so besser, je mehr sinnvolle Adressierungsarten sie hat, und hier glänzt der 68000 ganz besonders. Dieser Luxus macht die Sache zwar auch etwas schwierig, denn das alles will gelernt sein, und hier liegt auch die Barriere für die Kollegen, die es gewohnt sind, mit den wenigen (und primitiven) Adressierungsarten der »8-Bitter« auszukommen. Andererseits, wenn Sie das Thema beherrschen, dann beherrschen Sie auch den 68000.

Um zu zeigen, worum es geht: Im Beispiel von eben

```
MOVE.L  A3,A4
```

wurde der Inhalt des Registers A3 nach A4 kopiert. Schreibe ich hingegen

```
MOVE.L  (A3),(A4)
```

heißt das, daß die Inhalte der Register als Adressen zu sehen sind. Hat zum Beispiel im Moment des Befehls A3 den Wert 4711 und A4 ist gleich 5711, dann wird ein Langwort von Adresse 4711 (da startend und Byte für Byte) nach Adresse 5711 kopiert.

Wir haben nun schon zwei Adressierungsarten kennengelernt, nämlich »Register direkt« (MOVE.L A3,A4) und »Register indirekt« (MOVE.L (A3),(A4)).

Um noch eine Stufe höher zu gehen, schauen wir uns »Adreß-Register indirekt mit Postinkrement« an. Das sieht zum Beispiel so aus:

```
MOVE.W (AØ)+,DØ
```

Im Klartext: Kopiere das Wort, auf das A0 zeigt nach D0 und erhöhe danach (post) A0 um 2. Zwei deshalb, weil ein Wort 2 Byte hat. Bitte merken: Der 68000 ist eine Byte-Maschine, jede Adresse zeigt auf 1 Byte. »MOVE. (A0)+,D0« würde ein Langwort kopieren und danach A0 um 4 inkrementieren. Die nächste Variante wäre »Adreßregister indirekt mit Predekrement«. Ein Beispiel:

```
MOVE.L  DØ,-(A5)
```

In diesem Fall wird vorab (pre) 4 (Langwort hat 4 Byte) von A5 subtrahiert, dann wird D0 dahin kopiert, wohin A5 nun zeigt. Das hatten wir doch schon mal? Sie erinnern sich an den Stack aus Kapitel 2! Daten werden auf den Stack gebracht, indem man den Stackpointer (SP) erniedrigt und dann die Daten auf die Adresse kopiert, auf die SP zeigt.

Daten werden vom Stack geholt, indem man sie von der Adresse holt, auf die SP zeigt und dann SP erhöht. Das wäre dann unser schon bekanntes

```
MOVE.L (A5)+,DØ
```

Tatsächlich kann man so jedes Adreßregister als Stackpointer einsetzen.

Die Besonderheit des Registers A7, das auch in vielen Assemblern SP heißt, liegt darin, daß dieses Register auch durch Befehle wie JSR (Jump to Subroutine) und RTS

(Return) angesprochen wird. Man kann das aber auch mit anderen Registern erledigen, zum Beispiel kann man anstatt RTS auch schreiben

```
MOVE.L  (A7)+,AØ
JMP     (AØ)
```

Der MOVE-Befehl holt die Return-Adresse vom Stack in das Register A0, danach erfolgt ein Sprung (Jump) zur Adresse, auf die A0 nun zeigt. Sie sagen, warum der Umstand, ein RTS ist doch viel einfacher! Recht haben Sie, aber trotzdem werden Sie diese Lösung in Programmen sehen, die zum Beispiel von BASIC aus aufgerufen werden und zwar so:

```
MOVE.L  (A7)+, 4711
```

viele andere Befehle

```
MOVE.L  4711,AØ
JMP  (AØ)
```

Mit dem ersten MOVE-Befehl wird die Return-Adresse auf einen sicheren Platz in den RAM geholt (4711 ist hier nur symbolisch gemeint). Man sagt auch, die Return-Adresse wird gerettet. Wenn irgend etwas schief geht, kann ich dann immer noch mit Hilfe dieser Adresse zu BASIC zurück, egal wo der Stackpointer gerade steht. Nach diesem Ausflug in die Praxis, der einmal andeuten sollte, wofür man verschiedenartige Adressierungsarten braucht, wieder zurück zur Theorie. Bild 3.3 zeigt eine Liste aller Adressierungsarten und noch etwas mehr.

Zuerst notieren Sie bitte nur, daß eine Adresse aus mehreren Angaben zusammengesetzt sein kann. Die CPU errechnet daraus die endgültige Adresse, auch effektive Adresse (ea) genannt. Wie Sie schon wissen, belegt das Befehlswort 16 Bit. Die vier höherwertigsten davon beschreiben den Befehl an sich. Die übrigen 12 teilen sich in zwei Gruppen von 6 Bit, die die Adressierungsart von Ziel und Quelle (so vorhanden) angeben. Die 6 Bit je Operand wiederum teilen sich in zwei Gruppen von 3 Bit, die eine Gruppe heißt Modus, die zweite Register. Mit 3 Bit sind die Zahlen 0 bis 7 darstellbar, deshalb gibt es auch die Register A0—A7 bzw. D0—D7. Es gibt aber mehr als sieben Adressierungsarten, was damit erreicht wird, daß nicht bei jedem Adressiermodus alle Register erlaubt sind. Überhaupt, und das ist wichtig zu wissen, sind bestimmte Adressierungsarten nicht für den Quell- und (gleichzeitig) den Zieloperanden erlaubt und außerdem auch nicht für jeden Befehl. Zu diesem Thema finden Sie mehr Informationen im Anhang. Im Bild 3.3 sind Modus und Register als Binärzahlen dargestellt. Steht dort »An« oder »Dn«, können Sie dafür %000 bis %111 (dezimal 0 bis 7) einsetzen.

Adressierungsart	Kürzel	Modus	Register
Datenregister direkt	Dn	000	Dn
Adreßregister direkt	An	001	An
Adreßregister indirekt (ARI)	(An)	010	An
ARI mit Postinkrement	(An)+	011	An
ARI mit Predekrement	—(An)	100	An
ARI mit Adreßdistanz	d16(An)	101	An
wie vor plus Index	d8(An,Rn)	110	An
Absolut kurz	$XXXX	111	000
Absolut lang	$XXXXXXXX	111	001
PC-Relativ mit Adr.-Distanz	d16(PC)	111	010
PC-Relativ mit Adr.-Distanz plus Index	d8(PC,Rn)	111	011
Konstante, Statusregister	#, SR,CCR	111	100

Bild 3.3: Liste aller Adressierungsarten

3.6 Adressierungsarten im Detail

Wie Sie nun ganz richtig erkannt haben, geht aus dem Befehlswort und den darin codierten Adressierungsarten auch hervor, wie viele Worte der Befehl im Speicher belegt. Register-Register-Adressierung (zum Beispiel MOVE A0,A1) kommt mit einem Wort aus, geben Sie hingegen eine absolute Adresse an, kommt mindestens noch ein Wort hinzu. Das heißt, in den 16 Bit des Befehlswortes stecken alle Informationen, die die CPU braucht, um den Befehl zu decodieren.

So ähnlich arbeiten auch sogenannte Disassembler, das sind Programme, die aus dem Maschinencode wieder den Klartext der Assembler-Sprache bilden. Solange Sie aber ein solches Programm nicht schreiben wollen, können (und sollten) Ihnen diese Bitmuster herzlich egal sein. Es gibt viele Hacker, die den Hex-Code (die Maschinensprache) der »8-Bitter« lesen können wie andere Leute die Zeitung. Diese Übung ist beim 68000 aussichtslos, also sehen wir die Sache von der praktischen Anwendung her, nämlich alle Adressierungsarten an je einem Beispiel.

3.6.1 Register direkt

Eines der Register wird direkt angesprochen. Beispiel:

```
CLR  DØ                (Lösche DØ).
```

3.6.2 Adreßregister indirekt (ARI)

Der Inhalt des Registers ist eine Adresse, auf diese wirkt die Operation. Beispiel:

```
MOVE (AØ),DØ.          ARI bitte merken!
```

Das Wort, dessen Adresse in A0 steht, wird nach D0 kopiert.

3.6.3 ARI mit Postinkrement

Wirkt wie ARI, nur wird anschließend das Register inkrementiert. Beispiele:

```
MOVE.B   (AØ)+,DØ   ;Kopie,dann AØ=AØ+1
MOVE.W   (AØ)+,DØ   ;Kopie,dann AØ=AØ+2
MOVE.L   (AØ)+,DØ   ;Kopie,dann AØ=AØ+4
```

3.6.4 ARI mit Predekrement

Wie oben, nur wird das Register vor der Operation erniedrigt. Beispiel:

```
MOVE -(AØ), DØ      ;AØ=AØ-2, dann Kopie
```

3.6.5 ARI mit Adreßdistanz

Die effektive Adresse ist die Summe von Inhalt des Registers plus Adreßdistanz. Die Adreßdistanz ist eine vorzeichenbehaftete 16-Bit-Zahl im Bereich -32668..32767. Beispiel:

```
MOVE -1ØØ(AO), DØ
```

Wäre A0=500, würde das Wort von Adresse 400 nach D0 kopiert. Diese Adressierungsart sollten Sie sich besonders gut merken!!! Beim Amiga werden wir sie sehr häufig brauchen.

3.6.5.1 ARI mit Adreßdistanz und Index

Nun wird es kompliziert. Zuerst: Die Adreßdistanz ist jetzt nur noch eine vorzeichenbehaftete 8-Bit-Zahl im Bereich von -128..127. Nun darf aber noch ein weiteres Register angegeben werden. Ein Beispiel:

```
MOVE  1ØØ(AØ,DØ),4711
```

100 ist die Adreßdistanz, A0 enthält die Basisadresse, in D0 steht der Index. Alle drei werden addiert. Die Summe ist eine Adresse, das Wort (Byte, Langwort), das da steht, wird ins Ziel (hier Adresse 4711) kopiert.

Beim Index darf auch ein anderer Typ angegeben werden, also auch D0.B oder D0.L wären erlaubt, bei Adreßregistern als Index natürlich nur An.W und An.L ($0<n<7$). Auch der Index ist vorzeichenbehaftet, womit er im Falle Langwort im Bereich von 2 Giga-Byte liegen muß (wenn wir die mal hätten). Dieser Befehl ist ideal für die Abarbeitung von Tabellen und Arrays. Oft wird dabei die Adreßdistanz nicht benötigt (die Laufvariable steht im Indexregister), weshalb man oft die Form von zum Beispiel »0(A3,D4.L)« sieht.

3.6.6 Absolute Adressierung

Dies ist der einfachste Fall. Beispiel:

```
MOVE  4711,5713
```

Das Wort von Adresse 4711/12 wird auf Adresse 5713/14 kopiert. Die CPU unterscheidet dabei noch zwischen kurz und lang (Adreßbereich nur 64 Kbyte oder die vollen 16 Mbyte des 68000). Praktisch merken Sie den Unterschied kaum (die lange Adresse erfordert mehr Bytes in der Befehlslänge und ist etwas langsamer).

3.6.7 Konstanten-Adressierung

Auch wieder etwas ganz Einfaches. Um eine Konstante zu bewegen, brauchen Sie nur das Zeichen # vorzusetzen. Um zum Beispiel das ASCII-Zeichen A in das Register D0 zu laden, schreiben Sie

```
      MOVE  #65,DØ
oder  MOVE  #'A',DØ
```

3.6.8 PC-relative Adressierung

Da muß ich etwas ausholen, damit Sie dieses Feature auch würdigen können. Sobald Sie in einem Assembler-Programm eine absolute Adresse angeben, ist das Programm an einen Ort im Speicher gebunden. Auch zum Beispiel »(A0)« (indirekt) ist in diesem Sinn absolut, denn vorher mußten Sie A0 mit einer Adresse versorgen.

Der von Ihnen geplante Adreßbereich kann aber schon belegt sein, also muß Ihr Programm auch an einem anderen Ort laufen können. Dazu gibt es zwei Möglichkeiten. Erstens, das Programm ist verschiebbar (relokatibel). Dafür sorgt der Assembler, in-

dem er mit dem Programm eine Tabelle aller absoluten Adressen abspeichert. Der Lader (oder das Programm selbst oder eine Utility) kann dann diese Adressen korrigieren, indem sie die Differenz zwischen geplanter und tatsächlicher Startadresse auf alle absoluten Adressen laut Tabelle addieren. Die zweite Möglichkeit ist, das Programm lageunabhängig (Position Independend) zu schreiben. In einem solchen Programm dürfen dann eben keine absoluten Adressen vorkommen, und genau da hilft der 68000 mit der PC-relativen Adressierung. Dabei wird die Adresse gerechnet als aktueller Stand des PC+Offset. Offset ist auch hier wieder auf -32768..32767 begrenzt. Beispiel:

```
MOVE    100(PC),D0
```

PC-relativ mit Adreßdistanz und Index

Hier gilt sinngemäß das für »ARI mit Adreßdistanz und Index« Gesagte, nur daß die Basisadresse hier PC+2 ist. Beispiel:

```
MOVE    100(PC,A0.W),D0
```

Das wär's vorerst an Theorie. Es fehlt zwar noch allerlei, aber das wird an passender Stelle anhand praktischer Beispiele erläutert. Im nächsten Kapitel kommen die ersten Listings. Außerdem müssen wir uns um die Bedienung von Editor, Assembler, Linker und Batch-Prozessor kümmern.

Zum DOS wäre auch noch einiges zu sagen, schließlich wollen wir die Räder nicht neu erfinden, sondern alles, was da schon eingebaut ist, kräftig nutzen.

Kapitel 4

Ganz schnell zur Praxis

Hier geht es um Exec und DOS,

die Bedienung des Assemblers

und natürlich die ersten Listings.

4.1 Ein Schnellkurs in Sachen DOS

Das Betriebssystem des Amiga besteht grob vereinfacht aus drei Teilen, nämlich aus

DOS
Intuition
Exec

Die Aufgabe eines jeden OS (Operating System) ist es, die Verbindung des Computers mit der Außenwelt herzustellen. Dinge, wie Zeichen von der Tastatur lesen, Zeichen auf dem Bildschirm darstellen oder Dateien von einer Diskette lesen, sind typische Aufgaben des OS.

Nun kann der Amiga bekanntlich wie ein Standard-Computer bedient werden (Sie tun das, wenn Sie im CLI sind) oder über die grafische Benutzeroberfläche »Workbench«. Immer noch ganz grob (genauer behandeln wir das später) kann man nun sagen:

Standard = DOS
Grafik = Intuition

Bliebe noch Exec, und das ist in unserem vereinfachten Modell primär für das Multitasking zuständig.

Das DOS (Disk Operating System) hat auch etwas mit Disketten zu tun, doch der Name untertreibt. Tatsächlich kann das DOS auch mit der Tastatur und dem Bildschirm umgehen, ja sogar typische Amiga-Windows öffnen, den Drucker bedienen und einiges mehr. Weil das DOS recht einfach zu handhaben ist, werden wir uns vorerst nur damit beschäftigen. Schließlich müssen Sie zuerst die Assembler-Programmierung an sich lernen, was schon Stoff genug ist. Sie gleichzeitig noch mit den komplizierten Teilen der Amiga-System-Software zu behelligen, verkneife ich mir deshalb (aber nur vorerst).

Aus Sicht des Programmierers ist das DOS eine Sammlung von Routinen (Unterprogrammen), die er alle benutzen darf. Einige davon, wie zum Beispiel Laden und Starten eines Anwenderprogramms, sind dem »Normalverbraucher« zugänglich, alle nur dem Assembler-Programmierer.

Jedes dieser Unterprogramme startet natürlich bei einer bestimmten Adresse, und demnach könnte man so ein Programm in der Form »JSR Adresse« aufrufen. Praktisch tut man das nicht, denn dann würde jede Änderung im OS dazu führen, daß sich einige oder alle Adressen verschieben und somit alle »alten« Programme nur noch Makulatur wären.

4.2 Aufruf von DOS-Routinen

Um also von absoluten Adressen unabhängig zu sein, arbeiten die meisten OS nach diesem Schema:

Alle Unterprogramme erhalten eine Nummer, Funktionsnummer genannt. Im OS steht eine Tabelle, in der notiert ist, welche Adresse zu jeder Funktionsnummer gehört. Das OS hat nun eine Routine, deren Adresse sich nie ändert. Das ist der Dispatcher. Um ein Unterprogramm aufzurufen, übergibt man dem Dispatcher die Funktionsnummer. Dieser berechnet danach (und mit Hilfe der Tabelle) die Adresse der Routine und ruft sie auf.

Der Amiga macht das etwas raffinierter und damit zukunftssicherer. Der Nachteil der Standardmethode ist nämlich, daß man sehr schlecht neue Routinen hinzufügen kann (Tabelle steht im ROM). Beim Amiga stehen die Tabellen im ROM oder RAM oder auf der Diskette. Die Tabellen sind ein Teil der sogenannten Libraries (Bibliotheken).

Libraries: Schlüssel zum Amiga

Eine Library ist vereinfacht ausgedrückt eine Sammlung von Unterprogrammen mit einer zugehörigen Tabelle (je Unterprogramm ein Eintrag). Für jeden Zweck (zum Beispiel DOS, Intuition, Grafik) gibt es eine eigene Library. Will man eine Funktion einer Library benutzen, muß man die Library mit »OpenLibrary« öffnen. Diese Funktion gibt einen Zeiger auf den Beginn (die Startadresse) der Tabelle zurück. Um nun ein Unterprogramm aufrufen zu können, muß man die Startadresse der Tabelle angeben und ein sogenanntes Offset, das die Differenz zwischen Startadresse und zugehörigem Tabellenplatz ist.

»Gemanagt« wird das Ganze vom sogenannten Library-Manager (ein Teil von Exec). Der Manager weiß, ob sich eine Library schon im ROM oder RAM befindet. Wenn nicht, versucht er, die Library von der Diskette zu laden. Klappt das nicht (Library ist nicht auf der Diskette oder Speicher ist schon voll), gibt er Null als Adresse zurück.

Der Umstand hat noch einen Grund: Wir haben einen Amiga und der unterscheidet sich von seinen Konkurrenten auch durch sein Multitasking-System. Das heißt vereinfacht (kommt auch noch genauer), daß quasi gleichzeitig verschiedene Tasks (Programme) eine Library benutzen können. Der erste Task wird die Library notfalls von der Diskette in den RAM laden (genauer: das Laden veranlassen). Öffnen weitere Tasks dieselbe Library, wird der Manager nur noch die Adresse an diese Tasks melden. Daraus folgt: Eine Library darf erst wieder aus dem Speicher gelöscht werden, wenn der letzte Task gesagt hat, daß er sie nicht mehr braucht.

Dafür gibt es die Funktion »CloseLibrary«. Jeder Task (also jedes Programm, das Sie schreiben) muß deshalb alle Libraries, die er geöffnet hat, auch wieder schließen. Andernfalls könnte bald der Speicher knapp werden.

4.3 Aufbau eines Assembler-Programms

Jedes Assembler-Programm besteht aus den Feldern Marke, Befehle, Operanden (falls vorhanden) und Kommentar. Hier ein Muster:

```
Marke     Befehl    Operand(en)  Kommentar
-------------------------------------------------
Start     clr       dØ           ;Lösche Register
          move      dØ,d1        ;Befehl mit 2 Operanden
weiter                           ;nur Marke in der Zeile
          rts                    ;kein Operand
```

Der Kommentar muß nicht sein, er trägt aber zur Lesbarkeit bei. Je nach Assembler muß er mit einem Semikolon oder Stern beginnen, bei manchen Assemblern reicht auch der Platz (Kommentarfeld). Steht der Kommentar allein in einer Zeile, muß »;« oder »*« sein.

Die Marke (Label) wird nur in einigen Fällen gebraucht. Sie kann auch allein in einer Zeile stehen, sie wirkt aber immer auf die nächste Zeile mit einem Befehl. In manchen Assemblern muß der Marke ein Doppelpunkt folgen, aber nur dann, wenn sie im Markenfeld steht, nicht wenn sie angesprochen wird.

Ausgenommen die Sonderfälle »nur Marke« oder »nur Kommentar« muß ein Befehl in einer Zeile stehen, und, so vorhanden, auch dessen Operand(en). Die einzelnen Felder müssen durch mindestens eine Leerstelle voneinander getrennt sein. Meistens benutzt man die Tabulatortaste (8-ter Abstand). Nehmen Sie aber einen Texteditor, der dafür Blanks erzeugt.

4.4 Das erste Listing: Ausgabe eines Strings

Nun zu unserem ersten Programm laut Bild 4.1a:

```
* A1_Met   Mein erstes Programm   !!! Metacomco-Version !!!
* --------------------------------------------------------

        INCLUDE "libraries/dos_lib.i"
        XREF    _DOSBase
        XREF    _SysBase
        XREF    _LVOOpenLibrary
        XREF    _LVOCloseLibrary
        XDEF    _main
_main   move.l  #dosname,a1            ;Name der DOS-Lib
        moveq   #0,d0                  ;Version egal
        move.l  _SysBase,a6            ;Basis Exec
        jsr     _LVOOpenLibrary(a6)    ;DOS-Lib oeffnen
        tst.l   d0                     ;Fehler?
        beq     fini                   ;wenn Fehler, Ende
        move.l  d0,_DOSBase            ;Zeiger merken

* Ausgabe-Handle ermitteln:

        move.l  _DOSBase,a6            ;DOS-Funktion rufen
        jsr     _LVOOutput(a6)         ;Hole Output-Handle
        move.l  d0,d4                  ;und in d4 merken

* nun Text ausgeben:

        move.l  d4,d1                  ;Ausgabe-Handle
        move.l  #string,d2             ;Addresse Text
        moveq   #20,d3                 ;Laenge Text
        move.l  _DOSBase,a6            ;Basis DOS
        jsr     _LVOWrite(a6)          ;Funktion "Schreiben"

* Zum Schluss immer die Lib schliessen!

        move.l  _DOSBase,a1            ;Basis der Lib
        move.l  _SysBase,a6            ;Basis Exec
        jsr     _LVOCloseLibrary(a6)   ;Funktion "Schliessen"

fini    rts                            ;Return zum CLI

* Datenbereich:

dosname  dc.b   'dos.library',0
         cnop   0,2

string   dc.b   'Hallo lieber Leser!',10
         cnop   0,2

         end
```

Bild 4.1a: Ausgabe eines Strings (Metacomco-Assembler)

Zur Einführung stelle ich Ihnen nur das Listing dreimal vor, nämlich für die Assembler Metacomco, SEKA und DEVPAC. Alle weiteren Listings gelten für den DEVPAC-Assembler von HiSoft. Die Leser mit den anderen Assemblern sollten anhand der hier gegebenen Hinweise in der Lage sein, die Listings anzupassen. Speziell für die SEKA-Anwender sind die LVO-Tabellen im Anhang gedacht. Beginnen wir mit Metacomco, der nach der klassischen Methode arbeitet.

Das Programm soll schlicht aber herzlich »Hallo lieber Leser« auf den Schirm schreiben und dann zum CLI zurückkehren.

Vergessen wir vorerst den Overhead und betrachten das Listing ab der Zeile, die mit »_main« beginnt.

Auch für das simpelste Programm müssen wir eine Library öffnen. Dazu brauchen wir die Funktion OpenLibrary, die selbst in der Exec-Library zu finden ist. Wir brauchen aber auch immer zu jeder Funktion die Basisadresse der Library, und auch die erhält man mit OpenLibrary. Damit sich die Katze nun nicht in den Schwanz beißt, gibt es im Amiga eine feste Adresse (die einzige!), und das ist die Basis von Exec. Diese Adresse (4) hat den symbolischen Namen _AbsExecBase oder (hier benutzt) _SysBase.

Der Schlüssel steckt in diesen Zeilen:

```
        move.l  _SysBase,a6             ;Basis Exec
        jsr     _LVOOpenLibrary(a6)     ;DOS-Lib oeffnen
```

Im Klartext: Lade das Register a6 mit der Konstanten _SysBase. Dann springe zum Unterprogramm (JSR = Jump to Sub Routine), dessen Adresse sich aus der Konstanten _LVOOpenLibrary und dem Register a6 berechnet (Adressierungsart ARI mit Offset, siehe Kapitel 3).

Vorher müssen wir aber noch sagen, welche Library geöffnet werden soll. Dazu müssen zwei Parameter übergeben werden, nämlich der Name der Library und die Versionsnummer. Das erledigen die Zeilen

```
_main   move.l  #dosname,a1             ;Nane der DOS-Lib
        moveq   #0,d0                   ;Version egal
```

Die erste Zeile heißt: Kopiere (move) die Adresse von dosname in das Register a1. Das Doppelkreuz ist von immenser Bedeutung. Es heißt hier nämlich »Adresse von«. Vergessen Sie dieses Zeichen, gibt es einen bildschönen Absturz, denn dann heißt es »Inhalt von«. Bleibt noch die Versionsnummer: Es kann Libraries geben, die sich nicht im Namen aber in der Versionsnummer unterscheiden. Nur die Version 0 gibt es nie. Null ist reserviert für »nehme die erste (meistens die einzig vorhandene) Version«. Nach dem JSR kehrt das Unterprogramm zurück, und im Register d0 steht die Basis-

adresse der DOS-Library. Diese Adresse wird sofort in der Variablen _DOSBase gesichert.

Wir haben nun die Adresse der DOS-Library ermittelt und können damit arbeiten. Um einen Text ausgeben zu können, müssen wir zuerst wissen, wohin der Text geschrieben werden soll. Das kann eine Datei sein. Im Sinn von DOS ist aber auch das aktuelle Ausgabegerät eine Datei mit dem speziellen Namen Output. Vom Start des Amiga her (und solange wir nichts ändern) ist Output der Bildschirm (genauer: das CLI-Fenster). Immer noch: Für DOS ist das eine Datei, ein File, und der Zugriff auf Files geschieht über sogenannte Handles. Normalerweise öffnet man einen File mit Open. Nur ist ja unser Output-File schon offen, und deshalb gibt es eine Funktion mit dem Namen _LVOOutput, die die Handle von Output ermittelt. Genau das geschieht mit diesen Zeilen:

```
move.l  _DOSBase,a6           ;DOS-Funktion rufen
jsr     _LVOOutput(a6)        ;Hole Output-Handle
move.l  dØ,d4                 ;und in d4 merken
```

Wir rufen die DOS-Funktionen prinzipiell genauso auf wie die Exec-Funktionen. Der Unterschied ist nur, daß jetzt das Register a6 auf die Basis der DOS-Library (_DOSBase) zeigt. Auch die Konstante _LVOOutput ist woanders definiert (kommt noch). Wie alle Funktionen gibt auch _LVOOutput ihr Ergebnis in d0 zurück.

Da d0 ein Register ist, das auch andere Funktionen benutzen, retten wir es (kopieren es) in das Register d4.

Jedenfalls haben wir nun die Handle im Register d4 und können damit arbeiten.

Um auf einen File (oder ein Gerät) zu schreiben, braucht DOS diese Parameter:
- Handle in d1
- Adresse, ab der die Daten zu finden sind, in d2
- Anzahl der Daten-Bytes in d3

Schauen wir uns nun diese Zeilen an, so finden wir alles wieder:

```
move.l  d4,d1                 ;Ausgabe-Handle
move.l  #string,d2            ;Addresse Text
moveq   #2Ø,d3                ;Laenge Text
```

So vorbereitet, können wir die Funktion _LVOWrite aufrufen:

```
move.l  _DOSBase,a6           ;Basis DOS
jsr     _LVOWrite(a6)         ;Funktion "Schreiben"
```

Wie Sie sehen, ist das Prinzip immer dasselbe:

```
move.l  Basis_Adresse,a6
jsr     Offset(a6)
```

Nach dieser Methode wird denn auch zum Schluß des Programms die DOS-Library wieder geschlossen.

Im Datenbereich finden Sie nun einige Assembler-Direktiven. Wichtig ist die Assembler-Direktive »dc.b«. Bitte beachten Sie: Das ist eine Anweisung an den Assembler, kein 68000-Befehl. »dc« heißt »define constant« (definiere Konstante), »dc.b« heißt dann Konstante vom Typ Byte.

```
dosname dc.b    'dos.library',Ø
        cnop    Ø,2

string  dc.b    'Hallo lieber Leser!',1Ø
        cnop    Ø,2
```

»string« ist ein Label, und die ganze Anweisung an den Assembler lautet nun: Setze ab (symbolischer) Adresse string die Zeichenfolge »Hallo...« ein. Ja, und »Hallo« wollen wir nun ausdrucken. Dazu benötigen wir einen Zeiger, der auf »Hallo« (genauer: zuerst auf H) zeigt. Dazu ernennen wir das Register d2. Damit d2 mit der Adresse von string geladen wird, schreiben wir die Zeile

```
move.l   #string,d2              ;Addresse Text
```

Nochmals: »dc.b« heißt »definiere Konstante« und zwar hier vom Typ Byte. Im Operandenfeld stehen dann die Bytes. Diese können Sie einzeln eingeben (dc.b 100,33,20) als Text in Hochkommas oder wie hier gemischt. Der Name einer Library muß mit einem Null-Byte abgeschlossen sein, daher auch die Null am Ende des ersten »dc.b«.

Beim zweiten String ist kein Null-Byte erforderlich, weil die Write-Funktion die Länge als Parameter erwartet. Die 10 am Ende dieses Textes ist der ASCII-Code für »neue Zeile«. Damit sorge ich nur dafür, daß nach dem Programmlauf der CLI-Prompt (1>) auf einer neuen Zeile startet.

»cnop 0,2« ist eine andere Form für »even«, die ich gewählt habe, weil sie zumindest der HiSoft- und der Metacomco-Assembler gleichermaßen verstehen (bei SEKA heißt das ALIGN). Sie erinnern sich: Man sollte Texte immer auf einer Wortgrenze beginnen lassen. Noch sicherer (und bei manchen Funktionen Pflicht) ist eine Langwortgrenze (cnop 0,4).

Nun bliebe noch die Frage zu klären, wo die Konstanten wie zum Beispiel _LVOOpenLibrary herkommen. Schauen wir uns dazu das SEKA-Listing in Bild 4.1.b an.

```
* A1_Seka  Mein erstes Programm !!! Seka-Version !!!
* ---------------------------------------------------

SysBase:         equ    4                   ;Basis von Exec
LVOOpenLibrary:  equ    -552                ;Library oeffnen
LVOCloseLibrary: equ    -414                ;Library schliessen

LVOOutput:       equ    -60                 ;DOS: Output-Handle holen
LVOWrite:        equ    -48                 ;     Ausgabe

*DOS/Lib oeffnen:

main:   move.l   #dosname,a1                ;Name der DOS-Lib
        moveq    #0,d0                      ;Version egal
        move.l   SysBase,a6                 ;Basis Exec
        jsr      LVOOpenLibrary(a6)         ;DOS-Lib oeffnen
        tst.l    d0                         ;Fehler?
        beq      fini                       ;wenn Fehler, Ende
        move.l   d0,DOSBase                 ;Zeiger merken

* Ausgabe-Handle ermitteln:

        move.l   DOSBase,a6                 ;DOS-Funktion rufen
        jsr      LVOOutput(a6)              ;Hole Output-Handle
        move.l   d0,d4                      ;und in d4 merken

* nun Text ausgeben:

        move.l   d4,d1                      ;Ausgabe-Handle
        move.l   #string,d2                 ;Addresse Text
        moveq    #20,d3                     ;Laenge Text
        move.l   DOSBase,a6                 ;Basis DOS
        jsr      LVOWrite(a6)               ;Funktion "Schreiben"

* Zum Schluss immer die Lib schliessen!

        move.l   DOSBase,a1                 ;Basis der Lib
        move.l   SysBase,a6                 ;Basis Exec
        jsr      LVOCloseLibrary(a6)        ;Funktion "Schliessen"

fini:   rts                                 ;Return zum CLI

* Datenbereich:

DOSBase: dc.l    0
         align   4
dosname: dc      'dos.library',0
         align 4

string:  dc      'Hallo lieber Leser!',10
         align 4
```

Bild 4.1.b: Das Hallo-Programm in der SEKA-Version

Hier finden Sie in den ersten Zeilen sogenannte Equates. Auch das sind Assembler-Direktiven.

```
SysBase:   equ    4
```

definiert die Konstante SysBase mit dem Wert 4. Demnach sind diese Schreibweisen gleichwertig:

```
move.l   SysBase,a6
move.l   4,a6
```

Man sollte jedoch generell die erste Form wählen. Sowohl beim Metacomco- als auch DEVPAC-Assembler gibt es Include-Files (Textmodule), in denen diese Konstanten definiert sind. Inzwischen dürfte Ihnen auch aufgefallen sein, daß sämtliche Offsets mit _LVO beginnen. LVO heißt Library Vector Offset. Bei SEKA gibt es da außer der DOS-Lib (und das ist nur $^1/_{16}$ von allem) nichts. Die Unterschiede von SEKA zum Standard zeigt man am besten anhand der DEVPAC-Lösung von Bild 4.1.c.

```
        opt     l-                          ;nicht linken!

* A1_Dev   Mein erstes Programm !!! DevPac-Version !!!
* -----------------------------------------------------

_SysBase          equ    4                  ;Basis von Exec
_LVOOpenLibrary   equ    -552               ;Library oeffnen
_LVOCloseLibrary  equ    -414               ;Library schliessen

_LVOOutput        equ    -60                ;DOS: Output-Handle holen
_LVOWrite         equ    -48                ;     Ausgabe

*DOS/Lib oeffnen:

_main   move.l  #dosname,a1                 ;Name der DOS-Lib
        moveq   #0,d0                       ;Version egal
        move.l  _SysBase,a6                 ;Basis Exec
        jsr     _LVOOpenLibrary(a6)         ;DOS-Lib oeffnen
        tst.l   d0                          ;Fehler?
        beq     fini                        ;wenn Fehler, Ende
        move.l  d0,_DOSBase                 ;Zeiger merken

* Ausgabe-Handle ermitteln:

        move.l  _DOSBase,a6                 ;DOS-Funktion rufen
        jsr     _LVOOutput(a6)              ;Hole Output-Handle
        move.l  d0,d4                       ;und in d4 merken
```

```
* nun Text ausgeben:

        move.l  d4,d1                   ;Ausgabe-Handle
        move.l  #string,d2              ;Address Text
        moveq   #20,d3                  ;Laenge Text
        move.l  _DOSBase,a6             ;Basis DOS
        jsr     _LVOWrite(a6)           ;Funktion "Schreiben"

* Zum Schluss immer die Lib schliessen!

        move.l  _DOSBase,a1             ;Basis der Lib
        move.l  _SysBase,a6             ;Basis Exec
        jsr     _LVOCloseLibrary(a6)    ;Funktion "Schliessen"

fini    rts                             ;Return zum CLI

* Datenbereich:

_DOSBase dc.l   0

dosname  dc.b   'dos.library',0
         cnop   0,2

string   dc.b   'Hallo lieber Leser!',10
         cnop   0,2
```

Bild 4.1.c: Das Hallo-Programm in der DEVPAC-Version

Bei Metacomco und DEVPAC beginnen alle Konstanten mit dem Unterstrich. Dieser ist bei SEKA als erstes Zeichen nicht erlaubt. »dc.b« kennt SEKA nicht, man muß das b weglassen.

Für DOS_Base habe ich mit »dc.l 0« diese Adresse mit dem Langwort 0 belegt. An sich brauche ich aber nur Speicherplatz für diese Variable, wofür man normalerweise »ds.l 1« schreibt (definiere Speicher für 1 Langwort). Da SEKA diese Direktive nicht kennt, bin ich auf »dc.l« ausgewichen.

In der Metacomco-Lösung fehlt das ganz. Dafür sehen Sie am Anfang des Listings sehr oft XDEF. Das heißt, diese Dinge sind extern definiert (genauer: im File »amiga.lib«). Bei Metacomco ist nach dem Assemblieren noch ein Linkerlauf erforderlich. Dabei werden auch die externen Referenzen behandelt. SEKA und DEVPAC kommen ohne Linkerlauf aus, sprich können mit dem Assembler schon lauffähige Programme erzeugen.

4.5 Assemblieren und Linken

Wenn Sie nun dieses Programm mit einem Editor eingetippt haben, geht die Arbeit erst los. Speichern Sie den Text zum Beispiel unter dem Namen TEST.S und kehren zum Desktop zurück. Nun folgen mehrere Schritte, die von Ihrem Assembler-Paket abhängen. Lesen Sie bitte in Ihrem Handbuch nach.

Bei SEKA und HiSoft ist die Sache ganz einfach. Geben Sie das A-Kommando beziehungsweise Amiga-A bei HiSoft. Bei letzterem sollten Sie noch darauf achten, daß »nicht linkbarer Code« gewählt wurde, womit ein sofort ausführbares Programm entsteht. Dazu können Sie im Listing »opt l-« als erste Zeile eingeben. Bitte beachten Sie: Es muß wirklich die erste Zeile sein, auch eine Leerzeile davor ist nicht erlaubt. Bequemer ist es vielleicht, das Install-Programm laufen zu lassen und »opt l-« damit fest voreinzustellen.

Auch bei Metacomco müssen Sie zuerst assemblieren. Hier ist dann noch ein Linkerlauf erforderlich. Da beides viel Tipperei bedeutet, schreiben Sie am besten diesen Batch-File und speichern ihn unter dem Namen make im s-Directory.

```
.key file/a
c/assem <file>.s  -o <file>.o  -c s -i include
c/alink <file>.o to <file> library lib/amiga.lib
```

Tippen Sie nun

```
        execute make test        (ohne .s!!!)
```

Das heißt, assembliere »test.s« und schreibe das Ergebnis in den Code-File »test.o«. Lassen Sie das »-o test.o« weg, wird nur assembliert, aber kein Objekt-File erzeugt. Das geht sehr schnell und empfiehlt sich, wenn man ein Programm nur auf Fehlerfreiheit testen will.

Haben Sie keinen Fehler gemacht (der Assembler hat nicht gemeckert), dann beginnt das »Linken«. Mein privater heißer Tip: Besorgen Sie sich »BLINK«, was zum DEVPAC gehört, aber als Public Domain gilt. Dieser Linker ist deutlich schneller als ALINK, speziell die »Amiga-Gedenkminute« (der Linker tut anscheinend eine ganze Weile gar nichts) fehlt.

4.6 Eingabe von Strings

Nun mögen ja Texte, die wir ausgeben, ganz informativ sein, aber im allgemeinen erwarten wir wohl auch Eingaben von der Tastatur. Mit Bild 4.2 gehen wir deshalb einen kleinen Schritt weiter.

```
        opt     l-                      ;nicht linken!

* A2    Mein zweites Programm

_SysBase         equ    4               ;Basis von Exec
_LVOOpenLibrary  equ    -552            ;Library oeffnen
_LVOCloseLibrary equ    -414            ;Library schliessen

_LVOOutput       equ    -60             ;DOS: Output-Handle holen
_LVOWrite        equ    -48             ;     Ausgabe
_LVORead         equ    -42
_LVOInput        equ    -54

*DOS/Lib oeffnen:

_main   move.l  #dosname,a1             ;Name der DOS-Lib
        moveq   #0,d0                   ;Version egal
        move.l  _SysBase,a6             ;Basis Exec
        jsr     _LVOOpenLibrary(a6)     ;DOS-Lib oeffnen
        tst.l   d0                      ;Fehler?
        beq     fini                    ;wenn Fehler, Ende
        move.l  d0,a6                   ;Zeiger merken

* Ausgabe-Handle ermitteln:

        jsr     _LVOOutput(a6)          ;Hole Output-Handle
        move.l  d0,d4                   ;und in d4 merken

        move.l  d4,d1                   ;nun Text ausgeben
        move.l  #string,d2              ;wie gehabt
        moveq   #12,d3
        jsr     _LVOWrite(a6)

* Nun lese von der Tastatur:

        jsr     _LVOInput(a6)           ;Hole Input-Handle
        move.l  d0,d1                   ;und in d1 kopieren
        move.l  #buffer,d2              ;Adresse des Puffers
        moveq   #80,d3                  ;erlaube 80 Zeichen
        jsr     _LVORead(a6)            ;und lese
        move.l  d0,len
```

```
* nun Inhalt buffer ausgeben:

        move.l  d4,d1               ;Ausgabe-Handle
        move.l  #buffer,d2          ;Address Text
        move.l  len,d3              ;Laenge Text
        jsr     _LVOWrite(a6)       ;Funktion "Schreiben"

* Zum Schluss immer die Lib schliessen!

        move.l  a6,a1               ;DOS-Lib-Basis
        move.l  _SysBase,a6         ;Basis Exec
        jsr     _LVOCloseLibrary(a6) ;Funktion "Schliessen"

fini    rts                         ;Return zum CLI

* Datenbereich:

dosname  dc.b   'dos.library',0
         cnop   0,2

string   dc.b   'Enter Text: '
         cnop   0,2
buffer   ds.b   80
len      ds.l   1
```

Bild 4.2: Eingabe von Strings

Bis hin zur Zeile »* nun lese von der Tastatur:« hat sich gegenüber dem ersten Listing nichts geändert. Der ausgegebene Text heißt jetzt »Enter Text: «, und an dieser Stelle soll der User lostippen. Damit wir wissen, woher die Eingaben kommen, müssen wir zuerst die Handle der Tastatur kennen. Analog zur Output-Handle ermitteln wir diese mit der Funktion _LVOInput.

Haben wir die Input-Handle, können wir damit _LVORead aufrufen. Diese Funktion verhält sich sonst wie _LVOWrite, nur daß wir hier nicht die Adresse eines Textes übergeben, sondern die Adresse eines Puffers, in dem die Eingabe abgelegt werden soll. Mit

```
        buffer   ds.b    80
```

weist man den Assembler an, nun eine Lücke von 80 Byte zu lassen. Mit »ds.w 40« oder »ds.l 20« hätte ich dasselbe erreicht. Auf die gleiche Art wird ein Langwort reserviert, in dem wir die Länge der Eingabe ablegen können. Die Funktion _LVORead ist sehr flexibel. Die Längenangabe, mit der sie aufgerufen wird, ist immer das Maximum. Geben Sie weniger ein (Ende mit Return-Taste), steht nach dem JSR die Ist-Länge im

Register d0. Ich kopiere dann d0 in die Variable len. Das muß hier zwar nicht sein, sollte aber einmal gezeigt werden. Anschließend wird nämlich len wieder in das Register d3 kopiert, um die Ihnen schon bekannte Funktion _LVOWrite aufzurufen, die jetzt allerdings den Pufferinhalt ausgibt.

4.7 Schleifen

In diesem Abschnitt geht es um Schleifen. Vorab möchte ich allerdings etwas zur Rationalisierung unserer Arbeit tun. Wir haben nun schon zweimal diese Folge eingetippt:

```
_SysBase         equ    4                     ;Basis von Exec
_LVOOpenLibrary  equ    -552                  ;Library oeffnen
_LVOCloseLibrary equ    -414                  ;Library schliessen

_LVOOutput       equ    -60                   ;DOS: Output-Handle holen
_LVOWrite        equ    -48                   ;     Ausgabe
_LVORead         equ    -42
_LVOInput        equ    -54

*DOS/Lib oeffnen:

_main   move.l  #dosname,a1                   ;Name der DOS-Lib
        moveq   #0,d0                         ;Version egal
        move.l  _SysBase,a6                   ;Basis Exec
        jsr     _LVOOpenLibrary(a6)           ;DOS-Lib oeffnen
        tst.l   d0                            ;Fehler?
        beq     fini                          ;wenn Fehler, Ende
        move.l  d0,a6                         ;Zeiger merken
```

Bild 4.3: Ein Include-File, der noch oft gebraucht wird

Speichern Sie den Text von Bild 4.3 mittels der Blockfunktion Ihres Editors unter dem Namen »OpenDos.i« ab. In den folgenden Programmen reicht dann eine Include-Anweisung, und der Assembler wird diesen Text automatisch einziehen. Bei Metacomco muß der Dateiname in Anführungszeichen oder Hochkommas stehen. SEKA kann kein Include, dort müssen Sie den Text mittels des R-Kommandos lesen.

Die folgende Aufgabe lautet: Es sollen die Buchstaben von A bis Z ausgegeben werden. Das lösen wir zuerst ganz primitiv und beschränken uns deshalb auch auf A bis D. Aber

auch dazu müssen die Zeichen in einem Puffer stehen; die Frage ist nur, wie wir sie da hinein bekommen.

In Bild 4.4 stelle ich ein Register auf den Beginn des Puffers, und zwar mit

```
        lea.l    buffer,aØ
```

Das heißt »Lade a0 mit der effektiven Adresse von buffer«. Das ».l« ist an sich überflüssig (Adressen sind immer lang), aber manche Assembler bestehen trotzdem darauf. Der Befehl ist von gleicher Wirkung wie

```
        move.l  #buffer,aØ
```

Mit der Anweisung

```
        move.b   #'A',(aØ)+
```

wird die Konstante A auf die Adresse geschrieben, auf die a0 zeigt (hier Beginn buffer), und anschließend wird a0 um eins hochgezählt. Ein beliebter Fehler ist übrigens, das ».b« wegzulassen. Die Assembler setzen dafür nämlich automatisch ».w« ein, womit in diesem Fall a0 um 2 erhöht wird.

Wie auch immer, a0 zeigt schon auf die nächste Adresse, und wir können somit das Spielchen fortsetzen.

```
        opt     l-                          ;nicht linken!

* A3   Mein drittes Programm

        include  OpenDos.i

* Ausgabe-Handle ermitteln:

        jsr     _LVOOutput(a6)              ;Hole Output-Handle
        move.l  dØ,d4                       ;und in d4 merken

* Puffer fuellen

        lea.l   buffer,aØ
        move.b  #'A',(aØ)+
        move.b  #'B',(aØ)+
        move.b  #'C',(aØ)+
        move.b  #'D',(aØ)+
        move.b  #1Ø,(aØ)
```

```
* nun Inhalt buffer ausgeben:

        move.l  d4,d1                   ;Ausgabe-Handle
        move.l  #buffer,d2              ;Address Text
        move.l  #5,d3                   ;Laenge Text
        jsr     _LVOWrite(a6)           ;Funktion "Schreiben"

* Zum Schluss immer die Lib schliessen!

        move.l  a6,a1                   ;DOS-Lib-Basis
        move.l  _SysBase,a6             ;Basis Exec
        jsr     _LVOCloseLibrary(a6)    ;Funktion "Schliessen"

fini    rts                             ;Return zum CLI

* Datenbereich:

dosname dc.b    'dos.library',0
        cnop    0,2

buffer  ds.b    80
```

Bild 4.4: Puffer mit Zeichen füllen. Lösung 1

4.7.1 Die DBcc-Schleife

In Bild 4.4 tun wir immer wieder das gleiche; Grund genug uns einer effektiveren Technik für Wiederholungen, nämlich den Schleifen zuzuwenden. Wir wollen die Buchstaben A bis Z drucken und zwar in der Art, wie man als BASIC-Programm schreiben würde:

```
10 FOR I= ASC("A") TO ASC("Z")
20 PRINT CHR$(I)
30 NEXT
```

Bild 4.5 bringt die Lösung:

```
        opt     l-                      ;nicht linken!

* A4    Mein viertes Programm

        include  OpenDos.i

* Ausgabe-Handle ermitteln:

        jsr     _LVOOutput(a6)          ;Hole Output-Handle
        move.l  dØ,d4                   ;und in d4 merken

* Puffer fuellen:

        lea.l   buffer,aØ
        move    #25,dØ                  ;Siehe Text
        move.b  #'A',d1

loop    move.b  d1,(aØ)+
        addq    #1,d1
        dbra    dØ,loop
        move.b  #1Ø,(aØ)                ;Neue Zeile

* nun Inhalt buffer ausgeben:

        move.l  d4,d1                   ;Ausgabe-Handle
        move.l  #buffer,d2              ;Address Text
        move.l  #27,d3                  ;Laenge Text
        jsr     _LVOWrite(a6)           ;Funktion "Schreiben"

* Zum Schluss immer die Lib schliessen!

        move.l  a6,a1                   ;DOS-Lib-Basis
        move.l  _SysBase,a6             ;Basis Exec
        jsr     _LVOCloseLibrary(a6)    ;Funktion "Schliessen"

fini    rts                             ;Return zum CLI

* Datenbereich:

dosname dc.b    'dos.library',Ø
        cnop    Ø,2

buffer  ds.b    8Ø
```

Bild 4.5: Drucken von A bis Z mit DBcc

Im Gegensatz zu den meisten Konkurrenten hat der 68000 einen Schleifenbefehl schon eingebaut, nämlich:

```
        DBcc    Dn,Marke
```

Das steht für »Decrement and Branch on Condition Code«. Wow, das ist ein Ding!

Also der Reihe nach: Mit dem DBcc-Befehl wird immer ein Datenregister angegeben, das kann D0 bis D7 sein; nennen wir es Dn. Vor dem Eintritt in die Schleife wird Dn ein Wert zugewiesen. In der Schleife, genau immer dann, wenn der DBcc-Befehl durchlaufen wird, wird Dn um eins dekrementiert. Solange Dn dabei nicht —1 wird, erfolgt ein Sprung zu »Marke«, ansonsten wird der nächstfolgende Befehl ausgeführt. Nun zum »cc«: Zusätzlich kann man nun noch vor dem DBcc-Befehl mit zum Beispiel einer CMP-Anweisung (compare = vergleiche) eine Bedingung testen und dann zum Beispiel sagen:

```
CMP     (AØ)+,DØ
DBeq    D1,Marke
```

In diesem Fall erfolgt der Sprung zu »Marke« nur dann, wenn die Bedingung »A(0) eq (equal = gleich) D0« erfüllt ist, ansonsten wird die Schleife beendet. Man kann es auch so sehen: Die Schleife wird durchlaufen, solange die cc-Bedingung nicht erfüllt ist, aber höchstens solange, wie der Zähler noch nicht auf —1 ist. Die Kürzel für »cc« sind die gleichen, wie beim bcc-Befehl. Zum Beispiel gibt es BEQ (Branch if Equal) und DBEQ (Decrement and Branch if Equal). Die Einzelheiten zu allen »cc« finden Sie im nächsten Kapitel, machen wir aber erst einmal mit der Praxis weiter. Häufig interessiert nämlich die Bedingung überhaupt nicht, man will nur zählen. In diesem Fall sagt man einfach

```
DBRA,
```

was Decrement and Branch Allways (springe immer) heißt, natürlich nur solange der Zähler nicht abgelaufen ist. Häufig sieht man auch »DBF«, wobei F für »False« (Falsch) steht; das ist nur eine andere Schreibweise. Gute Assembler akzeptieren sowohl »RA« als auch »F«. Nun können wir uns dem Listing von Bild 4.5 zuwenden. Wir wollten die 26 Buchstaben von A bis Z drucken. Weil der Zähler d0 aber immer bis —1 läuft, initialisiere ich ihn mit 25, siehe erste Zeile. Den Code für Buchstaben halte ich im Register d1, das also zuerst mit »A« geladen wird.

Bei »Loop« geht es nun los. Wie gehabt, packen wir ein Zeichen mittels »(a0)+« in den Puffer. Doch nun das Neue: Mit »addq #1,d1« wird d1 inkrementiert, aus dem A wird also ein B (dann aus dem B ein C usw.). Die Arbeit leistet die nächste Zeile:

```
dbra   dØ,loop
```

heißt: dekrementiere d0. Wenn es dann noch nicht —1 ist, springe zu »Loop«, ansonsten nächster Befehl. Hier ginge es also im Fall von —1 bei der Ausgabe weiter.

4.8 Die Kommandozeile

Sozusagen als Einlage möchte ich Ihnen noch ein Programm vorstellen, das einen Text ausgibt. Diesmal aber einen Text, der im Programm nirgends definiert ist. Sie rufen bekanntlich ein Programm unter CLI auf, indem Sie nur seinen Namen eintippen. Sie können aber dem Namen nach einem Leerzeichen noch beliebigen Text folgen lassen. Diesen Text nennt man die Kommandozeile. Viele CLI-Kommandos arbeiten damit. Tippen Sie zum Beispiel

```
        cd dfØ:
```

so rufen Sie damit ein Programm namens cd auf und übergeben die Kommandozeile »df0:«.

Wie man das Kommando liest, zeigt Bild 4.6.

```
        opt     1-                      ;nicht linken!

* A5    Mein fuenftes Programm

* Immer zuerst Adresse und Laenge der Kommandozeile retten
        movem.l aØ/dØ,-(sp)

        include OpenDos.i

        jsr     _LVOOutput(a6)          ;Hole Output-Handle
        move.l  dØ,d1                   ;da soll sie hin

        movem.l (sp)+,aØ/dØ             ;Parameter zurueck

        move.l  aØ,d2                   ;Adresse Kommandozeile
        move.l  dØ,d3                   ;Laenge
        jsr     _LVOWrite(a6)           ;Funktion "Schreiben"

* Zum Schluss immer die Lib schliessen!

        move.l  a6,a1                   ;DOS-Lib-Basis
        move.l  _SysBase,a6             ;Basis Exec
        jsr     _LVOCloseLibrary(a6);Funktion "Schliessen"

fini    rts                             ;Return zum CLI

* Datenbereich:

dosname dc.b    'dos.library',Ø
```

Bild 4.6: Lesen der Kommandozeile

Das DOS speichert die Kommandozeile im RAM und stellt das Register a0 auf die Anfangsadresse. In d0 wird die Länge notiert. Da prinzipiell die Register d0, d1, a0 und a1 Scratch (Schmierpapier) sind (jede Routine kann sie ändern), muß ein Programm, das die Kommandozeile benötigt, zuerst die Register a0 und d0 retten.

Normalerweise speichert man beide Register in Variablen ab, ich möchte Ihnen aber eine andere Möglichkeit vorstellen, und das wäre der Stack.

Der 68000 kann mit einem einzigen Befehl alle oder einige Register auf den Stack bringen oder von dort holen. Mit

```
        movem.l  aØ/dØ,-(sp)
```

werden a0 und d0 auf dem Stack abgelegt. Es sind beliebige Kombinationen wie »a0/a3/a5/d1/d6« erlaubt oder Listen wie »d0-d7/a0-a4« (d0 bis d7, a0 bis a4). Wichtig ist, daß Sie dann später mit dem analogen Befehl die Daten auch wieder vom Stack holen. In unserem Fall geschieht das mit

```
        movem.l (sp)+,aØ/dØ        ;Parameter zurueck
```

Danach müssen wir nur noch die Register in die Parameter-Register der Write-Funktion kopieren und können die Kommandozeile ausgeben. Später werde ich Ihnen ein Programm vorstellen, das die Kommandozeile untersucht und daraus Aktionen ableitet.

4.9 Unterprogramme

Das Programm von Bild 4.7 soll fragen »Wie heißt Du?«. Der User gibt dann einen Text ein (ich hoffe, seinen Namen) , und das Programm antwortet dann »Guten Tag, lieber Name«.

Alles was wir dafür brauchen, kennen Sie schon (Eingabe und Ausgabe von Strings), nur langsam wird die Sache lästig. Wir müssen nämlich dreimal einen Text ausgeben. Jedesmal die ganze Sequenz dafür zu schreiben, ist zwar möglich, aber besser erledigt man das mit Unterprogrammen. Das Problem bei Unterprogrammen ist die Parameterübergabe. Wenn ich alle drei Parameter der Write-Funktion übergebe (drei Befehle) und dann mein Unterprogramm »print« aufrufe, habe ich nichts gewonnen. Ich könnte dann auch gleich die drei Parameter an Write übergeben und diese DOS-Routine rufen. Daher folgende Vereinbarungen:

1. Die Output-Handle ist bekannt (steht in d4)
2. Es wird nur die Adresse des Textes übergeben. Im Text ist die Länge »versteckt«:

Wie man das löst, zeigt Bild 4.7:

```
        opt     l-                      ;nicht linken!

* A6    Mein sechstes Programm

        include OpenDos.i

        jsr     _LVOOutput(a6)          ;Hole Output-Handle
        move.l  dØ,d4
        lea.l   msg1,aØ                 ;frage nach Namen
        bsr     print

        jsr     _LVOInput(a6)           ;Hole Input-Handle
        move.l  dØ,d1                   ;und arbeite damit
        lea.l   buffer,a2               ;Zeiger auf Puffer
        move.l  a2,d2                   ;an Read uebergeben
        addq.l  #1,d2                   ;Laengen-Byte skippen
        move.l  #79,d3                  ;so lang darf Name sein
        jsr     _LVORead(a6)            ;Ist-Laenge in dØ
        addq.l  #1,dØ                   ;verlaengern
        move.b  dØ,(a2)                 ;und eintragen
        move.b  #'!',-1(a2,dØ.l)        ;! in Puffer
        move.b  #1Ø,Ø(a2,dØ.l)          ;und noch neue Zeile

        lea.l   msg2,aØ                 ;sage Guten Tag
        bsr     print

        move.l  a2,aØ                   ;drucke Namen
        bsr     print

* Zum Schluss immer die Lib schliessen!

        move.l  a6,a1                   ;DOS-Lib-Basis
        move.l  _SysBase,a6             ;Basis Exec
        jsr     _LVOCloseLibrary(a6)    ;Funktion "Schliessen"

fini    rts                             ;Return zum CLI

print   clr.l   d3
        move.b  (aØ)+,d3                ;Laenge
        move.l  d4,d1                   ;Output-Handle
        move.l  aØ,d2                   ;Adresse
        jsr     _LVOWrite(a6)           ;Funktion "Schreiben"
        rts

* Datenbereich:

dosname dc.b    'dos.library',Ø
```

```
        cnop    Ø,2

msg1    dc.b    16,'Wie heisst Du?  '
        cnop    Ø,2
msg2    dc.b    19,1Ø,'Guten Tag, lieber '
        cnop    Ø,2
buffer  ds.b    8Ø
```

Bild 4.7: Unterprogramme in der Praxis

Der Trick steckt in den letzten Zeilen. Die ersten Bytes der Strings msg1 und msg2 halten die Länge des folgenden Textes. Da Strings in Pascal so abgelegt werden, spricht man auch von Pascal-Strings.

Der Aufruf des Unterprogramms erfolgt nach diesem Schema:

```
        lea.l   msg1,aØ                 ;frage nach Namen
        bsr     print
```

Das heißt, nur die Adresse des jeweiligen Strings wird übergeben. BSR heißt »Branch to Sub Routine« (Verzweige zum Unterprogramm). Der Unterschied zu JSR besteht darin, daß BSR auf eine Sprungweite von +/- 32 Kbyte begrenzt ist, während JSR für den vollen Adreßbereich des 68000 (16 Mbyte) gilt. Sie müssen nicht BSR nehmen, das spart nur etwas Code und Zeit. Nun zum Unterprogramm selbst:

```
print   clr.l   d3
        move.b  (aØ)+,d                 ;Laenge
        move.l  d4,d1                   ;Output-Handle
        move.l  aØ,d2                   ;Adresse
        jsr     _LVOWrite(a6)           ;Funktion "Schreiben"
        rts
```

In der zweiten Zeile steckt des Pudels Kern. Das Längen-Byte wird in das Register d3 kopiert (da erwartet es Write). Das kleine »+« stellt gleichzeitig a0 auf den Beginn des Textes. Wir können dann getrost zwei Zeilen später a0 in d2 kopieren. Das klappt, nur der direkte Zugriff auf den Textbeginn geht schief (ungerade Adresse, Write muß diesen Fall abfangen). Nun zur ersten Zeile: Die Länge muß als Langwort übergeben werden, wir haben aber nur ein Byte. Unser Problem:

Stehen in einem Register die vier Byte

```
        B3 B2 B1 BØ
```

dann »moved«

```
move.b              B0
move oder move.w    B1, B0
move.l              B3, B2, B1 B0
```

Wenn wir nur ein Byte übertragen, bleibt der Rest von 3 unverändert. Damit können dann im Langwort d3 »Hausnummern« entstehen. Daher lösche ich vorher mit clr (clear = lösche, fülle mit Nullen) das Register. Übrigens: Wenn Sie schreiben

```
moveq     #1,d3
```

ist das O.K., weil moveq (move quick) automatisch die Konstante auf »long« erweitert. Die Konstante ist übrigens auf 8 Bit (-128 bis +127) begrenzt.

Hat man sich erst einmal auf ein bestimmtes Verfahren zur Parameterübergabe an Unterprogramme eingelassen, dann kann das unter Umständen recht gravierende Folgen haben. Um das zu zeigen, habe ich mir die Auflage erteilt, daß auch Texte, die mit Read gelesen wurden, mit der Print-Routine ausgegeben werden sollen.

Read liest bekanntlich in einen Puffer und zwar an sich ab dessen Beginn. Nun erwartet aber Print als erstes Zeichen in diesem Puffer die Länge des Textes. Das hat zur Folge:

```
lea.l    buffer,a2                  ;Zeiger auf Puffer
move.l   a2,d2                      ;an Read uebergeben
addq.l   #1,d2                      ;Laengen-Byte skippen
move.l   #79,d3                     ;so lang darf Name sein
jsr      _LVORead(a6)               ;Ist-Laenge in d0
move.b   d0,(a2)                    ;und eintragen
```

Die ersten beiden Zeilen sind noch die üblichen. Wir stellen a2 als Zeiger auf den Pufferbeginn und kopieren dann a2 nach d2, wo Read üblicherweise die Pufferadresse erwartet. Doch nun wird d2 um 1 erhöht. Damit zeigt d2 auf das zweite Byte im Puffer. Read wird also ab dieser Adresse den Puffer füllen, unser Längen-Byte bleibt frei. Nach dem JSR wird dann einfach die Ist-Länge in den Puffer kopiert, was

```
move.b  d0,(a2)
```

erledigt. Doch nun schauen Sie auf das Listing, da steht noch mehr. Der Grund: Nach dem Namen soll noch ein Ausrufungszeichen gedruckt werden. Dafür wird a) die Ist-Länge erhöht und b) das Ausrufungszeichen in den Puffer geschrieben. Dahinter soll dann noch wieder die 10 (neue Zeile) folgen. Daher:

```
move.b  #'!',-1(a2,d0.l)          ;! in Puffer
move.b  #10,0(a2,d0.l)            ;und noch neue Zeile
```

Falls Sie sich bisher noch nicht vorstellen konnten, was man mit »ARI mit Index und Offset« (siehe Kapitel 3) anfangen könnte, hier haben wir eine praktische Anwendung.

Zuerst stolpern Sie vielleicht über das »—1«. Dazu muß man wissen, daß die Read-Funktion auch die Return-Taste (ASCII-Code 10) als letztes Zeichen im Puffer ablegt und auch bei der Länge mitzählt.

```
(a2, dØ.1)
```

heißt: bilde die Adresse aus Summe von a2 + d0. Beginnt der Puffer zum Beispiel auf Adresse 1000, und haben wir die Zeichen ABC eingetippt, so stehen im Puffer:

Adresse	=	1000	1001	1002	1003
Zeichen	=	A	B	C	Return

Die Länge ist 4. Folglich ist 1000 + 4 (a2 + d0) = 1004.

Wir wollen aber das Ausrufungszeichen auf die Adresse 1003 bringen. Daher addieren wir noch das Offset von —1, sprich subtrahieren 1. Der nächste Befehl

```
move.b  #1Ø,Ø(a2,dØ.1)
```

addiert ein Offset von Null. Damit wird dann »#10« auf die Adresse 1004 geschrieben. Wir können die Null nicht weglassen. Die Syntax des Befehls will dort eine Konstante sehen. Übrigens findet man häufig diese Form mit dem Null-Offset, weil es meistens reicht, die Adresse nur aus den beiden Registern zu bilden.

4.10 Programmsegmente Text, Data und BSS

Häufig finden Sie in Listings nun noch diese Direktiven:

```
text
data
bss
```

Eventuell steht noch das Wort SECTION davor, für »text« trifft man auch »Code« an.

»data« ist die Anweisung an den Assembler, die folgenden Daten in das Datensegment des Programms zu packen. Dazu müssen Sie wissen: Ein Programm kann aus Segmenten bestehen. Das erste Segment heißt Text oder Code. Darin steht das eigentliche Programm. Sie können, in manchen Assemblern müssen Sie sogar, Ihr Programm mit dem Wort »text« starten.

Im Data-Segment stehen alle initialisierten Daten, also solche, die einen Wert haben, wie zum Beispiel unsere Texte. Im »bss« (block storage segment) werden Daten abgelegt, die erst während der Programmlaufzeit entstehen. Praktisch sind es nur reservierte Speicherbereiche (entstehen mit der DS-Direktive).

Wie gesagt, Sie können, Sie müssen nicht diese Sektionen bilden. Vorteilhaft ist das erst bei sehr großen Programmen, die damit dem Lader die Chance bieten, leichter noch passende freie Speicherbereiche für die einzelnen Segmente zu finden. DEVPAC unterstützt aber zur Zeit nur eine Sektion, SEKA keine.

Kapitel 5

Verzweigungen und Menü-Technik

In diesem Kapitel geht es um das »IF THEN« in Assembler und um das leidige, aber sehr notwendige Bit-Schieben.

Natürlich kommt auch wieder die Praxis an die Reihe.

Diesmal lernen wir den Einsatz der Funktionstasten und das Prinzip von »ON X GOSUB«.

5.1 IF THEN im Detail

Wir hatten es ja schon angedeutet, nun schauen wir es uns etwas genauer an, das »IF Bedingung THEN GOTO«. Prinzipiell funktioniert es so wie in den Hochsprachen: Man fragt eine Bedingung ab und verzweigt in Abhängigkeit vom Ergebnis. Der kleine Unterschied zu den Hochsprachen: Die Bedingung ist der Zustand einiger Bits im CCR (Condition Code Register), was wiederum Teil des Statusregisters ist. Wie dieses Register aussieht, zeigt das folgende Bild (Bild 5.1).

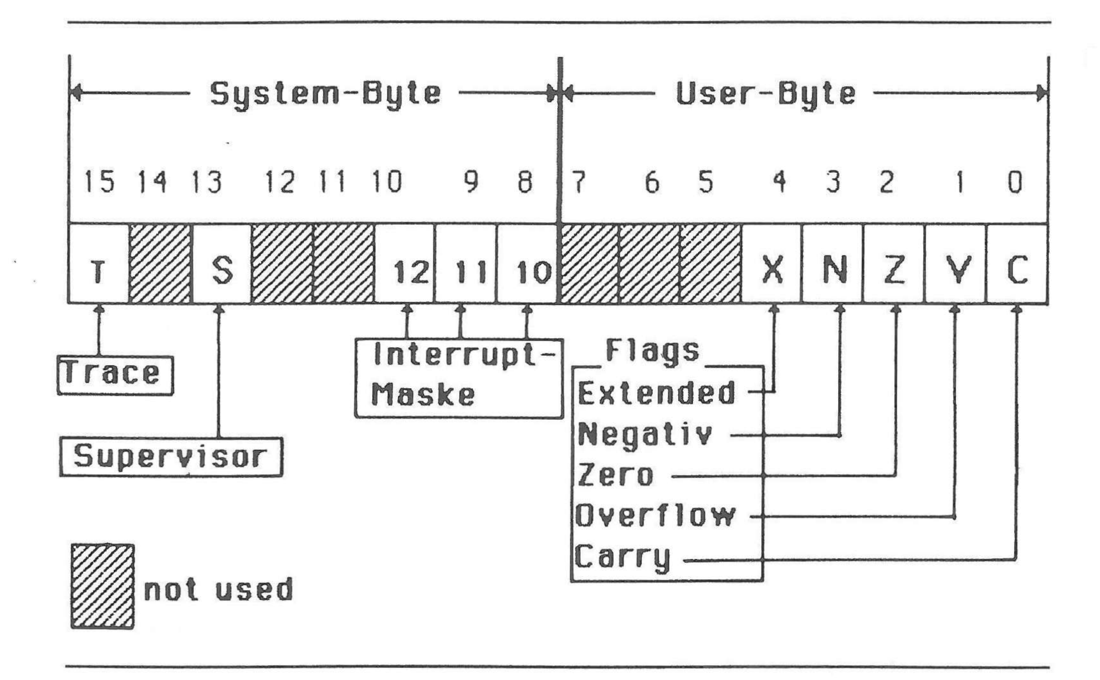


Bild 5.1: Das Statusregister des 68000

5.1.1 Das Statusregister

Sie sehen sofort, daß das Wort in ein System- und ein User-Byte unterteilt ist. Dazu wissen Sie schon, daß der 68000 zwei Betriebsarten kennt, nämlich Supervisor-Mode und User-Mode.

Der Supervisor-Mode sollte dem Betriebssystem vorbehalten bleiben, wir »User« arbeiten sicherheitshalber nur im User-Mode. Deshalb interessieren uns von den Bits (auch Flags genannt) auch zuerst nur X, N, Z, V und C.

5.1.2 Die Flags

Es gibt viele Befehle, die diese Flags beeinflussen, meistens jedoch sind es mathematische Operationen mit zwei Operanden, wobei der Quelloperand vom Zieloperanden subtrahiert wird. Wird dabei das Ergebnis negativ, setzt der 68000 das N-Flag (das Bit wird 1).

Entsteht ein Überlauf, wird das Overflow-Bit gesetzt, bei einem Übertrag (Addition) bzw. beim »Borgen« während der Subtraktion geht das Carry-Flag auf eins. Treten umgekehrt diese Zustände während der Operation nicht ein, werden die entsprechenden Flags auf Null gesetzt.

Vorerst nur als Info: T ist das Trace-Bit, S das Supervisor-Bit und i0, i1, i2 die Interrupt-Maske.

5.1.3 Die Abfrage der Flags

	Kürzel	Bedeutung	Deutsch
	CC	Carry Clear	Carry = 0
	CS	Carry Set	Carry = 1
	EQ	Equal	Z =1
	GE	Greater or Equal	> =
***	GT	Greater Than	>
	HI	Higher	>
***	LE	Less or Equal	<
	LS	Less or Same	< =
***	LT	Less Than	<
	MI	Minus	—
	NE	Not Equal	< >
	PL	Plus	+
***	VC	oVerflow Clear	V = 0
***	VS	oVerflow Set	V =1

Tabelle 5.1: Kürzel der »Condition Codes«

Die Flags selbst kann man zwar abfragen, aber das tut man im allgemeinen nicht. Stattdessen schreibt man einen Befehl, der die Flags beeinflußt (prüft) und dann sinngemäß »GOTO Adresse, wenn dieses Flag diesen Zustand hat«. Merken Sie sich bitte, daß GOTO hier Branch heißt, wofür man aber nur B schreibt. Ganz entscheidend ist nun, daß (im Gegensatz zu den »8-Bittern«) es auch Branch-Befehle gibt, die mehrere Flags auf einmal berücksichtigen. Noch ein Unterschied: Es gibt verschiedene Branch-Befehle für vorzeichenlose und vorzeichenbehaftete Zahlen (2er-Komplement). Natür-

lich gibt es auch Befehle, die nur auf ein Bit reagieren. Tabelle 5.1 bringt eine Übersicht.

Die in der Tabelle mit *** gekennzeichneten Operatoren gelten nur für Zahlen im sogenannten 2er-Komplement-Format, das sind diese, bei denen das höchstwertige Bit als Vorzeichen dient. Die Befehle fangen immer mit B (wie Branch) an, gefolgt von zwei Buchstaben, die die Kurzform für die Bedingung sind. Wenn Sie nun zum Beispiel BEQ (springe wenn gleich) schreiben, dann hängt es ausschließlich vom Z-Flag ab, ob der Befehl ausgeführt wird, oder nicht.

Das Z-Flag kann aber durch einen mehr oder weniger weit vor dem BEQ liegenden Befehl beeinflußt worden sein. Wenn Sie nun genau wissen, welcher Befehl das Z-Flag wie beeinflußt, dann könnten Sie das Risiko eingehen. Sicherer ist es auf jeden Fall, direkt vor dem BEQ einen Befehl zu schreiben, der das prüft. Wenn ich zum Beispiel springen will, wenn das Register D0=0 ist, dann schreibe ich:

```
CMP  #0,D0
BEQ  Marke
```

Der CMP-Befehl subtrahiert den Quelloperanden vom Zieloperanden, ändert je nach Ergebnis die Flags, schreibt aber nicht das Ergebnis ins Ziel. Das heißt, der Vergleichsbefehl wirkt auf die Flags wie eine Subtraktion. Das müssen sich die armen Kollegen mit den »8-Bittern« immer vor Augen führen, wenn Sie danach einzelne Flags (mit je einem Befehl) testen. Sie haben es besser. Sie dürfen zum Beispiel BGE (Springe wenn größer oder gleich) schreiben. Sie müssen sich nur dreierlei merken:

1. Diese Luxusbefehle sind nur direkt nach einem CMP korrekt wirksam.
2. Der zweite Operand wird mit dem ersten verglichen. Wenn ich zum Beispiel springen will, wenn D0 größer als 9 ist (D0>9), schreibe ich:

```
CMP  #9,D0
BGT  Marke
```

3. Man muß wissen, ob man die Operanden als vorzeichenbehaftete oder vorzeichenlose Zahlen vereinbart hat. Sinngemäß kann man die Kürzel auch im Zusammenhang mit DBcc anwenden, DBMI oder DBGT wären zwei Beispiele. BRA ist ein Sonderfall (springe immer); dem entspricht auch DBRA oder DBF.

5.2 Unser erstes Window

Mit Bild 5.2 komme ich wieder zur Praxis. Die Funktion Read legt einen Text Zeichen für Zeichen in einem Puffer ab. Nun will ich wissen, welche Taste welchen Code erzeugt. Dazu muß ich den Inhalt des Puffers ausdrucken und zwar in hex. Hauptaufgabe von Bild 5.1 ist somit die Ausgabe von Bytes in Form von je zwei Hex-Zeichen. Für ein

A wäre zum Beispiel 41 zu drucken. Nun muß ich Ihnen allerdings gleich verraten, daß wir mit der normalen Konsole, mit der wir bisher gearbeitet haben (CON:) zwar die Standardtasten erfassen, nicht aber die Sondertasten, wozu die Cursor-Steuerung und die Funktionstasten zählen. Um also in einem Programm auf die Funktionstasten reagieren zu können, müssen wir etwas Besonderes tun.

Zuerst dürfen wir nicht mehr die Standard-Handles Input und Output verwenden, sondern müssen selbst etwas für das I/O tun. Dazu bietet uns der Amiga die Auswahl unter einigen Devices (Geräten) an. Dazu zählen unter anderem:

PAR: Parallelschnittstelle
SER: Serielle Schnittstelle
CON: Konsole
RAW: Konsole

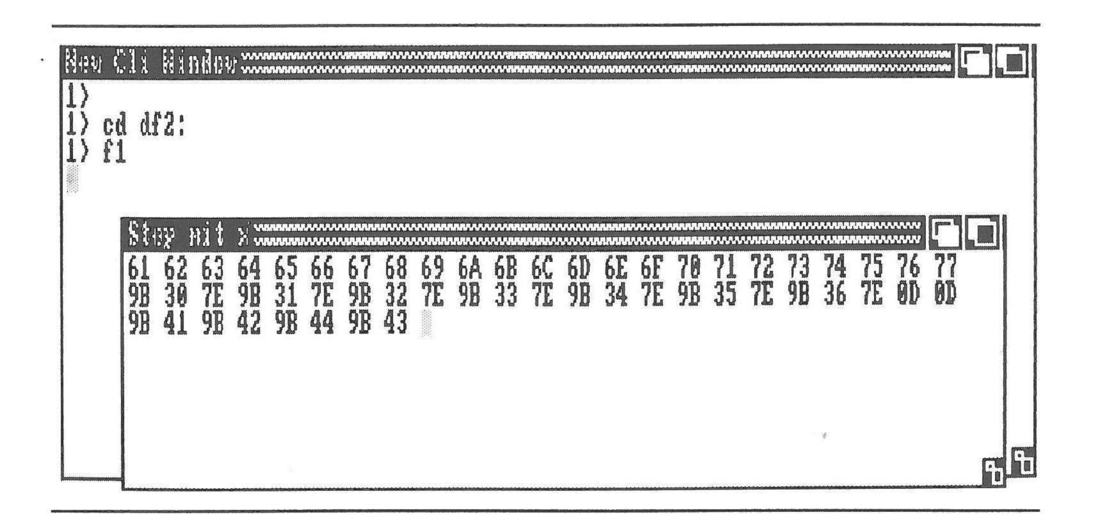


Bild 5.2 Unser erstes Window

Der Unterschied zwischen CON: und RAW: ist, daß nur lezteres alle Tasten (also auch die Sondertasten) behandelt. Der Nachteil von RAW: ist, daß alle Editierfunktionen ausgeschlossen, sprich vom Anwenderprogramm selbst zu stellen sind. RAW: ist das typische Device für Editor-Programme.

Bild 5.2 soll zeigen, was wir uns als Aufgabe vornehmen. Im CLI-Window erscheint ein neues (unser) Window. In diesem Fenster erscheinen Hex-Zahlen immer, wenn wir eine Taste tippen. Die Zahlen beschreiben die Codes der Tasten. In der ersten Zeile habe ich schlicht die Buchstaben von a bis w getippt. Sie sehen die zugehörigen ASCII-Codes von $61 bis $77 ($ heißt hex).

In der zweiten Zeile habe ich die Funktionstasten betätigt. Was Sie vielleicht nur erahnen können: Jede Taste erzeugt drei Zeichen und zwar:

[F1] 9B 30 7E
[F2] 9B 31 7E
[F3] 9B 32 7E
usw.

In der dritten Zeile stehen die Cursortasten. Diese generieren nur zwei Zeichen und zwar:

[Auf] 9B 41
[Ab] 9B 42
[Links] 9B 43
[Rechts] 9B 44

Allen Sondertasten ist also gemeinsam, daß sie eine Sequenz erzeugen, die mit $9B startet. Damit kann man recht einfach diese Tasten von allen anderen unterscheiden. Gleichzeitig ist das allerdings auch die Aufforderung, ein weiteres Zeichen auszuwerten, denn erst da beginnen die Unterschiede. Auf diese Feinheit verzichten wir vorerst und sagen einfach: alles anzeigen, was kommt. Nun aber zum Listing von Bild 5.3.

```
        opt      l-                          ;nicht linken!

* F1 Funktionstasten lesen


        include  OpenDos.i

_LVOOpen         equ  -30
_LVOClose        equ  -36

        move.l  #name,d1                     ;Name von RAW:
        move.l  #1005,d2                     ;Status = gibt es
        jsr      _LVOOpen(a6)                ;nun oeffnen
        move.l  d0,d5                        ;Handle merken
        tst.l   d0                           ;Fehler?
        beq     fini                         ;wenn ja, abbrechen
```

```
loop    move.l  d5,d1                   ;von RAW lesen
        move.l  #buffer,d2              ;in diesen Puffer
        move.l  #1,d3                   ;Länge, siehe Text
        jsr     _LVORead(a6)            ;Lesen aufrufen

        cmp.b   #'x',buffer             ;Zeichen = 'x' ?
        beq     fertig                  ;wenn ja

        move.l  buffer,d2               ;Zeichen nach d2
        lea.l   hbuf,aØ                 ;Ziel fuer Wandlung
        move.b  #' ',2(aØ)              ;Blank nach Hex
        bsr     hex                     ;in Hex-Zahl wandeln

* nun Hex-Zahl ausgeben:
        move.l  d5,d1                   ;auch Output im Window
        move.l  #hbuf,d2                ;Address Hex-String
        move.l  #3,d3                   ;Laenge
        jsr     _LVOWrite(a6)           ;Funktion "Schreiben"

        bra      loop                   ;auf ein Neues

fertig  move.l   d5,d1                  ;RAW wieder schliessen
        jsr     _LVOClose(a6)

* Zum Schluss immer die Lib schliessen!

        move.l  a6,a1                   ;DOS-Lib-Basis
        move.l  _SysBase,a6             ;Basis Exec
        jsr     _LVOCloseLibrary(a6)    ;Funktion "Schliessen"

fini    rts                             ;Return zum CLI

* Konvertiere d2.l in ASCII-String ab (aØ)
hex
        moveq   #2-1,d1                 ;nur fuer 2 Nibble (von 8)
next    rol.l   #4,d2                   ;hole 1 Nibble
        move.l  d2,d3                   ;nach d3 retten
        and.b   #$Øf,d3                 ;maskiere es
        add.b   #48,d3                  ;in ASCII wandeln
        cmp.b   #58,d3                  ;ist es >9 ?
        bcs     out                     ;wenn nicht
        addq.b  #7,d3                   ;sonst muss es A-F sein
out     move.b  d3,(aØ)+                ;1 Zeichen abspeichern
        dbra    d1,next                 ;next nibble
        rts
* Datenbereich:

dosname  dc.b   'dos.library',Ø
         cnop   Ø,2
name     dc.b   'RAW:4Ø/1ØØ/58Ø/8Ø/Stop mit x',Ø
         cnop   Ø,2
```

```
buffer  ds.b    8
        cnop    Ø,4

hbuf    ds.b    1Ø
```

Bild 5.3: So erfaßt man auch die Funktionstasten

Auch auf ein Gerät greifen wir wie auf einen File zu, müssen das Device also mit Open öffnen und später auch wieder schließen. Die zugehörigen DOS-LVOs finden Sie zu Anfang des Listings. Zum Öffnen müssen wir den Namen des Gerätes (des Files) übergeben und den Access Mode. In den Include-Files findet man die Equates

```
MODE_OLDFILE   EQU  1ØØ5
MODE_NEWFILE   EQU  1ØØ6
```

Gemeint ist damit, ob man auf eine schon existierende Datei zugreifen oder eine neue anlegen will. Da wir wissen (hoffen), daß RAW: existiert, wählen wir 1005. Unter »name« kommt nun die große Überraschung:

```
name    dc.b    'RAW:4Ø/1ØØ/58Ø/8Ø/Stop mit x',Ø
```

Das reicht, um das Gerät RAW: zu öffnen. Gleichzeitig schaffen wir damit ein Window mit diesen Eigenschaften:

40/100:	linke,obere Ecke (x,y in Bildschirmpunkten)
580:	Breite des Windows
80:	Höhe des Windows
Stop mit x:	Window-Titel

Die Funktion gibt wie üblich eine Handle zurück, die wir sowohl für die Eingabe als auch für die Ausgabe benutzen können. Das läuft nun wieder über die schon bekannten Funktionen Read und Write. Bei Read müssen Sie aufpassen. Auch wenn Sie als Länge 3 angeben, werden für die Funktionstasten 3 Reads erforderlich, sprich auch die Länge 1 tut es.

Nach dem Lesen eines Zeichens folgt

```
cmp.b   #'x',buffer              ;Zeichen = 'x' ?
beq     fertig                   ;wenn ja
```

und damit hätten wir unser erstes »IF THEN«. Im Klartext heißt das: Vergleiche (cmp = compare) die Konstante x mit dem ersten Byte im Puffer. Bei Übereinstimmung

springe zum Label »fertig« (branch if equal = springe wenn gleich). Soll auch heißen: Unser Programm läuft in einer Schleife, bis ein x eingegeben wird.

Die folgenden Zeilen bereiten die Hex-Wandlung vor. Das Unterprogramm hex erwartet ein Langwort in d2. Es legt das Ergebnis im Puffer hbuf ab.

5.3 Bit-Schieben muß sein

Das Unterprogramm zeigt recht gut, daß man in Assembler oft auf Bit-Ebene arbeiten sollte. Damit läßt sich zum Beispiel die Hex-Konvertierung wesentlich einfacher lösen, als mit der klassischen Methode (fortlaufende Division durch 16). Sie werden bei der Gelegenheit auch sehen, warum die Hex-Darstellung so vorteilhaft ist. Ein Beispiel: Das Langwort besteht aus den 4 Byte des Inhalts AA, BB, CC, DD. Ich sehe sofort, daß im höherwertigen Wort AABB und im niederwertigen CCDD steht. In dezimal (2864434397) dürfte das schwerfallen.

5.3.1 Ein Hex-Konverter

Das Problem der Routine »hex« ist nun, daß Sie tatsächlich ASCII-Zeichen ausgeben muß. Hat dort ein Digit den Wert Null, muß ich den ASCII-Code 48 (dezimal) ausgeben, um die »0« auf dem Schirm zu sehen. Das klappt ganz gut bis zur Neun (57), doch für 10 muß ich hex »A« drucken, und das hat den ASCII-Code 65, B hat 66 usw. Diese Lücke zwischen »9« und »A« müssen wir also überprüfen. Zweites Problem: Die Hex-Zahl sei $12345678 ($ heißt hex). $1 ist ein Nibble (Halb-Byte), das im Register 4 Bit belegt (0001). Natürlich muß ich $1 zuerst ausgeben, doch dafür muß ich $1 auf den Platz von $8 bringen, weil im Puffer das Zeichen (nach der Umwandlung in die ASCII-1) vorn stehen muß. Das Byte hat aber 8 Bit, 4 übertrage ich, die übrigen 4 Bit haben Werte, die nur stören, folglich muß ich sie ausblenden. Damit ergibt sich folgender Ablauf:

1. Das Nibble $1 auf den Platz von $8 bringen
2. Dort die übrigen 4 Bit des Bytes auf Null setzen
3. Das Nibble in ASCII wandeln
4. Das Zeichen im Puffer ablegen
5. Wiederhole 1. bis 4. mit den Nibbles $2, $3...$8
   Dazu eine Anmerkung: das Unterprogramm ist universell und kann auch Langworte konvertieren. Weil ich hier aber nur ein Byte (das höchstwertigste Byte!) (zwei Nibbles) ausgeben will, lasse ich die Schleife nur bis 2 − 1 laufen.

Schritt 1 wird mit dem ROL-Befehl erledigt (Rotate Left). Wir benutzen von ROL die Syntax

```
ROL #4,d2
```

Das heißt, rotiere den Inhalt von d2 um 4 Bit nach links. Und was heißt nun rotieren? Nehmen wir an, in den d2 stehen diese 32 Bit

vorher	1111 0000 0000 0000 0000 0000 0000 0000
nach ROL #4	0000 0000 0000 0000 0000 0000 0000 1111

Das heißt, die Bits werden nach links geschoben, und die Bits, die dann ganz links »herausfallen«, werden rechts wieder eingespeist. Anders sähe es beim ASL-Befehl (Shift left, schiebe nach links) aus. Da wird auch nach links geschoben. Rechts werden Nullen eingespeist und links fallen die Bits heraus. Nun, wir haben ROL gewählt, und damit unser Ziel erreicht. Das Nibble (so nennt man ein Halb-Byte), das vorher ganz links stand, steht nun ganz rechts.

Unsere Schleife soll zweimal durchlaufen werden, folglich muß ich vorab (wegen des -1) den Schleifenzähler D1 mit 1 (2-1) laden. Nun erfolgt das berühmte ROL. Danach steht das Nibble am richtigen Platz, aber ausgeben kann ich leider nicht diese 4 Bit alleine, ich brauche ein Byte für ein ASCII-Zeichen, also 8 Bit.

5.3.2 Die Sache mit den Masken

Die 4 Bit links von meinem Nibble haben aber einen Wert, und der muß weg (genauer: die 4 höherwertigen Bits des Bytes müssen 0000 werden). Das geschieht über die Maske $F mit »andi.b #$0F,d3«. Beispiel:

in d3 steht	1011	1010
AND Maske	0000	1111
ergibt	0000	1010

Das Verfahren lebt davon, daß logisch UND nur dann wahr (1) ergibt, wenn alle Eingangsgrößen wahr sind. In Assembler wirkt UND (and) Bit für Bit. Nachdem wir so den reinen Zahlenwert (0..15 dezimal) isoliert haben, beginnt die Konvertierung in ASCII. 0 bis 9 (als Zahl) kann direkt in 0 bis 9 (als ASCII-Zeichen) umgesetzt werden, das sind die ASCII-Codes 48 bis 57. Das heißt aber auch, 48 müssen wir mindestens addieren. Nun vergleichen wir D3 gegen 58 (also eine hex 10, die man als A ausgeben muß). Ist die Zahl <10 (also 0 bis 9) kann alles so bleiben, es geht zur Ausgabe. Andernfalls müssen wir noch die Lücke in der ASCII-Tabelle (65 für A-58=7) addieren.

An dieser Stelle ein Hinweis. Man sieht manchmal den Weg, daß erst geprüft wird, ob der Wert zwischen 0 und 9 liegt und dann im zweiten Test, ob es ein Wert zwischen 10 und 15 ist. Diese Methode ist O.K., wenn die Zeichen aus einer Eingabe stammen, wo

ja der User auch ungültige Zeichen (nicht 0..9, A...F) eintippen kann. Wir holen hier aber Zahlen aus einem Register, da so etwas nicht drin stehen kann!

Die Befehle CMP und BCS hatten wir ja schon. Warum aber BCS? Ja eigentlich nur, um Ihnen zu zeigen, was andere Leute so schreiben (und denken müssen), die von CPUs kommen, die nicht so komfortabel wie der 68000 sind. Ein Vergleich wirkt auf die Flags wie eine Subtraktion. Bei einem »Borgen« wird auch das Carry-Flag gesetzt. Das ist aber der Fall, wenn D3 > 58 ist.

5.4 Die Mehrfachverzweigung

Nun ist es ja üblich, daß auf den Druck einer Funktionstaste hin ein Programm etwas mehr tut, sprich eine Routine aufruft. Das können sehr komplizierte Programmteile sein; wir wollen aber das Prinzip üben in der Art von

```
IF   F1  THEN  GOTO ...
IF   F2   THEN GOTO ....
usw.
```

Die aufgerufene Routine soll auch nur »Hier ist F1, 2 usw.« ausgeben, und damit nicht so viel zu tippen ist, schon bei F4 enden. Bild 5.4 zeigt die Lösung.

```
        opt     l-                      ;nicht linken!

* F2 Funktionstasten lesen und agieren


       include  OpenDos.i

_LVOOpen        equ  -30
_LVOClose       equ  -36

        move.l  #name,d1                ;Name von RAW:
        move.l  #1005,d2                ;Status = gibt es
        jsr     _LVOOpen(a6)            ;nun oeffnen
        move.l  d0,d5                   ;Handle merken
        tst.l   d0                      ;Fehler?
        beq     fini                    ;wenn ja, abbrechen

        lea.l   buffer,a3               ;Adresse des Puffers

loop    jsr     GetKey                  ;Lese Taste
        cmp.b   #$9B,(a3)               ;Funktionstaste?
```

```
        bne     loop                    ;wenn nicht
        jsr     GetKey                  ;sonst lese Code

        cmp.b   #$3Ø,(a3)               ;Taste F1 ?
        beq     F1                      ;wenn ja
        cmp.b   #$31,(a3)               ;       F2 ?
        beq     F2                      ;wenn ja
        cmp.b   #$32,(a3)               ;       F3 ?
        beq     F3                      ;wenn ja
        cmp.b   #$33,(a3)               ;Taste F4
        beq     F4                      ;wenn ja, fertig
        bra     loop

F1      lea.l    f1_text,aØ             ;Adresse Text
        bsr      print                  ;drucken
        bra      loop                   ;auf ein Neues

F2      lea.l    f2_text,aØ             ;Adresse Text
        bsr      print                  ;drucken
        bra      loop                   ;auf ein Neues

F3      lea.l    f3_text,aØ                 ;Adresse Text
        bsr      print                  ;drucken
        bra      loop                   ;auf ein Neues

F4      move.l  d5,d1                   ;RAW wieder schliessen
        jsr     _LVOClose(a6)

* Zum Schluss immer die Lib schliessen!
        move.l  a6,a1                   ;DOS-Lib-Basis
        move.l  _SysBase,a6             ;Basis Exec
        jsr     _LVOCloseLibrary(a6)    ;Funktion "Schliessen"

fini    rts                             ;Return zum CLI

GetKey  move.l  d5,d1                   ;von RAW lesen
        move.l  a3,d2                   ;in diesen Puffer
        move.l  #1,d3                   ;1 Zeichen
        jsr     _LVORead(a6)            ;Lesen aufrufen
        rts

print   clr.l   d3
        move.b  (aØ)+,d3                ;Laenge
        move.l  d5,d1
        move.l  aØ,d2
        jsr     _LVOWrite(a6)            ;Funktion "Schreiben"
        rts

* Datenbereich:

dosname  dc.b   'dos.library',Ø
         cnop   Ø,2
```

```
name      dc.b    'RAW:4Ø/1ØØ/58Ø/8Ø/Stop mit F4',Ø
          cnop    Ø,2

f1_text   dc.b    8,'Hier F1',1Ø
          cnop    Ø,2
f2_text   dc.b    8,'Hier F2',1Ø
          cnop    Ø,2
f3_text   dc.b    8,'Hier F3',1Ø
          cnop    Ø,2

buffer    ds.b    8
          cnop    Ø,4

hbuf      ds.b    1Ø
```

Bild 5.4: Verzweigung mit vielen IF THEN

5.5 Lösung 1: Mit vielen IF THEN

Ich glaube, das Programm muß ich nicht mehr groß erklären, aber jedes Programm hat einen Sinn. Notfalls kann es nämlich als abschreckendes Beispiel dienen, und das soll es auch. Stellen Sie sich vor, ein Programm mit 100 Kommandos oder mehr würde so arbeiten. Das wäre ein Umstand und ein Spaghetti-Code!!

Neu ist nur, daß ich jetzt auf $9B prüfe und im Fall »keine Sondertaste« gleich auf das nächste Zeichen warte. Weil ich aber im Fall $9B noch eine Taste lesen muß, habe ich das Lesen in das Unterprogramm »GetKey« verlagert.

5.6 Lösung 2: ON X GOSUB in Assembler

Viel eleganter löst man so etwas mit einem »ON X GOSUB«. Dazu müssen wir zwar wieder einmal den Schwierigkeitsgrad etwas steigern, aber wenn Sie das gelernt haben, ist eigentlich schon fast alles gelaufen. Jedes Programm besteht nämlich aus einer Hauptschleife, in der es auf Kommandos wartet. Die Kommandos werden interpretiert und die passenden Unterprogramme aufgerufen. Danach geht es weiter in der Hauptschleife. In BASIC sähe das so aus:

```
1Ø    INPUT  KOMMANDO
2Ø    ON KOMMANDO GOSUB  1ØØ,2ØØ,3ØØ, ....
3Ø    GOTO 1Ø
```

Wie das in Assembler aussieht, zeigt Bild 5.5.

```
        opt      l-                         ;nicht linken!

* F3    ON Funktionstasten GOSUB

       include  OpenDos.i

_LVOOpen        equ  -30
_LVOClose       equ  -36

        move.l  #name,d1                    ;Name von RAW:
        move.l  #1005,d2                    ;Status = gibt es
        jsr     _LVOOpen(a6)                ;nun oeffnen
        move.l  d0,d5                       ;Handle merken
        tst.l   d0                          ;Fehler?
        beq     fini                        ;wenn ja, abbrechen

        lea.l   buffer,a3                   ;Adresse des Puffers

loop    jsr     GetKey                      ;Lese Taste
        cmp.b   #$9B,(a3)                   ;Funktionstaste?
        bne     loop                        ;wenn nicht
        jsr     GetKey                      ;sonst lese Code

        move.b  (a3),d0                     ;Code -> d0
        ext.w   d0                          ;auf Wort erweitern
        sub.w   #$30,d0                     ;Code in 0..3
        asl.w   #2,d0                       ;mal 4
        lea.l   table,a0                    ;Zeiger auf Tabelle
        move.l  0(a0,d0.w),a0               ;Adresse -> a0
        jsr     (a0)                        ;Routine aufrufen

        bra     loop                        ;bis F4 kommt

F1      lea.l    f1_text,a0                 ;Adresse Text
        bsr      print                      ;drucken
        rts

F2      lea.l    f2_text,a0                 ;Adresse Text
        bsr      print                      ;drucken
        rts

F3      lea.l    f3_text,a0                 ;Adresse Text
        bsr      print                      ;drucken
        rts

F4      move.l  (sp)+,d0                    ;Kill Return Adress
        move.l  d5,d1                       ;RAW wieder schliessen
```

```
        jsr     _LVOClose(a6)

* Zum Schluss immer die Lib schliessen!

        move.l  a6,a1                   ;DOS-Lib-Basis
        move.l  _SysBase,a6             ;Basis Exec
        jsr     _LVOCloseLibrary(a6)    ;Funktion "Schliessen"

fini    rts                             ;Return zum CLI

GetKey  move.l  d5,d1                   ;von RAW lesen
        move.l  a3,d2                   ;in diesen Puffer
        move.l  #1,d3                   ;1 Zeichen
        jsr     _LVORead(a6)            ;Lesen aufrufen
        rts

print   clr.l   d3
        move.b  (aØ)+,d3                ;Laenge
        move.l  d5,d1
        move.l  aØ,d2
        jsr     _LVOWrite(a6)           ;Funktion "Schreiben"
        rts

* Datenbereich:

table    dc.l   F1
         dc.l   F2
         dc.l   F3
         dc.l   F4

dosname  dc.b 'dos.library',Ø
         cnop   Ø,2
name     dc.b   'RAW:4Ø/1ØØ/58Ø/8Ø/Stop mit F4',Ø
         cnop   Ø,2

f1_text  dc.b   8,'Hier F1',1Ø
         cnop   Ø,2
f2_text  dc.b   8,'Hier F2',1Ø
         cnop   Ø,2
f3_text  dc.b   8,'Hier F3',1Ø
         cnop   Ø,2

buffer   ds.b   8
         cnop   Ø,4

hbuf     ds.b   1Ø
```

Bild 5.5: ON X GOSUB in Assembler

Den Anfang des Listings kennen Sie schon fast. Nur wird hier der Code der Funktionstasten (das Byte nach $9B) durch Subtraktion von $30 in die Zahlen 0 bis 9 gewandelt; so steht er dann im Register D0. Der Einfachheit halber bearbeiten wir auch hier nur die Funktionstasten [F1], [F2], [F3] und [F4], also 0, 1, 2 und 3 in D0. Bitte beachten Sie, daß alle Tests fehlen (es geht um das Prinzip!), daß bei Betätigung aller anderen Tasten das Programm also abstürzt. Wir wollen ein »ON D0 GOSUB« realisieren und brauchen dazu natürlich zuerst vier Unterprogramme, die hier wieder F1, F2, F3 und F4 heißen. Diese Unterprogramme sind trivial. Sie geben nur die Meldungen »hier ist ...« aus. Nun betrachten wir das Listing wieder von unten. Da steht die Marke »table«, und das, genau das, ist das Geheimnis unseres »ON X GOSUB«.

Für Mehrfachverzweigungen braucht man in Assembler eine Sprungtabelle. Diese Tabelle ist eine Liste mit den Adressen der Unterprogramme. Das Erstellen der Tabelle ist recht einfach. Schreiben Sie für jede Adresse

```
        dc.l    Label
```

wobei für Label die Marke (die symbolische Adresse) des jeweiligen Unterprogramms einzutragen ist. Wichtig ist die Reihenfolge. Wir haben hier die Zuordnung

Taste	Routine
F-Taste 1	F1
F-Taste 2	F2
F-Taste 3	F3

Die Reihenfolge der Tasten muß sich in der Reihenfolge der Einträge (so nennt man das) der Tabelle wiederholen. Die Unterprogramme selbst können in beliebiger Reihenfolge im Listing stehen. Es besteht also eine enge Zuordnung zwischen den Kommandos (hier den Funktionstasten) und der Sprungtabelle. Deshalb kann man aus den Kommandos die zugehörige Adresse berechnen. Eine Adresse belegt immer 4 Byte, also kann unsere Tabelle zum Beispiel so im Speicher stehen:

Label	Adresse
F1	1000
F2	1004
F3	1008

Unser Kommando (in D0) kann sein:

Wert	0,	1	oder	2
das mal 4 ergibt	0,	4	oder	8
und das plus 1000	1000,	1004	oder	1008

So einfach ist das also. Multiplizieren wir nun D0 mit 4. Dafür hat der 68000 natürlich auch einen speziellen Befehl (MULU), aber genau den nehmen wir hier nicht. Ein anständiger Assembler-Programmierer wird nämlich bei einer Multiplikation mit 2 oder 4 oder 8 oder 16 (Sie merken es: bei jeder 2er-Potenz) sofort hellhörig und greift auf einen Befehl zu, der das viel schneller erledigt. Hier heißt dieser Befehl

```
ASL     Arithmetic Shift Left
```

»ASL #1,d0« zum Beispiel schiebt alle Bits in D0 um eine Stelle nach links. Die Wirkung ist die gleiche wie die im Dezimalsystem, wo Sie durch Linksschieben der Zahlen (und Festhalten des Kommas) mit 10 multiplizieren. Hier sind wir aber im dualen Zahlensystem, womit sich nur eine Multiplikation mit 2 ergibt. Schieben wir aber um zwei Stellen, so hätten wir schon unser »mal 4«. Das Ergebnis müssen wir auf den Beginn der Tabelle addieren. Deren Startadresse beschaffen wir uns mit »lea table,a0«. Nun kommt ein ganz wilder Befehl, nämlich

```
move.l     Ø(aØ,dØ.w),aØ
```

Wir benutzen die Adressierungsart »ARI mit Index und Offset«. Nur »Index« gibt es leider nicht, also setzen wir das Offset zu Null. Demnach errechnet sich die effektive Adresse als die Summe von a0 und d0. Die müssen wir nun in das Zielregister »moven«, und da nehmen wir gleich wieder a0. So etwas ist beim 68000 erlaubt, und weil es schön »tricky« aussieht, schreibt es auch jeder so. A0 zeigt nun also auf die Adresse des zugehörigen Unterprogramms, und das können wir nun schlicht mit »jsr (a0)« aufrufen. Nach dem »jsr« kehrt das Programm zum dem »jsr« folgenden Befehl zurück. Der heißt »bra loop«, also wieder von vorn mit dem nächsten Kommando.

Die Ausnahme von dieser Regel finden Sie im Unterprogramm »F4«. Dieses Unterprogramm ist gar keins, es wird zwar mit JSR aufgerufen, es endet aber nicht mit RTS. Folglich müssen wir die noch auf dem Stack befindliche Return-Adresse entfernen. Die Amerikaner sagen dafür so schön »Kill Return Address«. Wir erledigen das hier mit dem Befehl »move.l (sp)+,d0«. Das ist zulässig, weil in diesem Fall D0 nicht mehr benötigt wird. Es hat sich aber auch die folgende Schreibweise eingebürgert:

```
move.l  (sp)+,(sp)
```

Damit wird die Return-Adresse vom Stack geholt und gleich wieder auf den Stack geschrieben. Wegen des »+« hat sich auch der Stackpointer geändert; der Zweck ist also erreicht. Natürlich wäre auch ein »addq.l #4,sp« korrekt, aber einige Leute haben so ihre speziellen Methoden, um eine Aktion wie »Kill Return Address« herauszustellen.

5.7 Lösung von CASE X OF

Ist doch ganz einfach, oder? Es ist Ihnen zu einfach? Nun gut, machen wir die Sache etwas komplizierter. Unser schöner Kommando-Interpreter hat einen Nachteil. Die Kommandos müssen in der Reihenfolge von F-Tasten-Codes auftreten. Auch andere Folgen, wie 1 bis 9 oder A bis M sind denkbar, es muß aber immer eine Folge sein. Sie wissen jetzt, warum manche Leute ein Menü anbieten der Art

1 = Eingabe
2 = Rechnen
3 = Stoppen

Das merkt sich schlecht, besser wäre doch

E = Eingabe
R = Rechnen
S = Stoppen

5.8 Arbeiten mit zwei Tabellen

Das Prinzip ist natürlich wieder ganz einfach. Es gibt zwei Tabellen. In der ersten Tabelle stehen die »Keys« (die erlaubten Tasten oder Kommandos), in der zweiten die Adressen der zugehörigen Routinen. So muß man doch nur den Key in der ersten Tabelle suchen und aus seiner Platznummer in dieser Tabelle einen Zeiger auf den richtigen Platz in der Adreßtabelle errechnen. Da kann man dann die Adresse herausholen und ab geht's.

Diesmal soll unser Programm aber wasserdicht sein. Folglich muß der Fall »nicht gefunden« abgefangen werden. Wir wollen auch den User nicht zwingen, immer die Shift-Taste zu betätigen. Deshalb sollen Groß- und Kleinbuchstaben gleich behandelt werden. Schließlich soll das Programm universell sein, sprich: Es muß mit minimalem Aufwand möglich sein, Funktionen hinzuzunehmen oder zu ändern.

Nun denn, hier ist Bild 5.6 mit dem Listing.

```
         opt     l-                           ;nicht linken!

* F4     CASE X OF

       include  OpenDos.i

_LVOOpen        equ  -30
_LVOClose       equ  -36

        move.l  #name,d1                      ;Name von RAW:
        move.l  #1005,d2                      ;Status = gibt es
        jsr      _LVOOpen(a6)                 ;nun oeffnen
        move.l  d0,d5                         ;Handle merken
        tst.l   d0                            ;Fehler?
        beq     fini                          ;wenn ja, abbrechen

        lea     buffer,a3                     ;Adresse des Puffers

loop    jsr     GetKey                        ;Lese Taste

* Key in Tabelle gueltiger Keys suchen
* ------------------------------------
        move.b  (a3),d0                       ;Code -> d0
        bclr    #5,d0                         ;force uppercase
        lea     keys,a0                       ;Tab. gueltige Keys
        move    #count,d1                     ;deren Anzahl
search  cmp.b   (a0)+,d0                      ;Key hier?
        dbeq    d1,search                     ;wenn nicht
        tst     d1                            ;Key gefunden?
        bmi     loop                          ;wenn nicht

* Rechne Adresse zu Key
* ---------------------
        neg     d1                            ;sub d1,#count
        add     #count,d1                     ;ergibt Platz-Nr. von Key
        lsl     #2,d1                         ;die mal 4
        lea     table,a0                      ;Zeiger auf Tabelle
        move.l  0(a0,d1),a0                   ;Adresse -> a0

        jsr     (a0)                          ;Routine aufrufen
        bra     loop                          ;bis F4 kommt

Eingabe lea      E_text,a0                    ;Adresse Text
        bsr      print                        ;drucken
        rts

Rechnen lea      R_text,a0                    ;Adresse Text
        bsr      print                        ;drucken
        rts

Stoppen move.l  (sp)+,d0                      ;Kill Return Adress
```

```
        move.l  d5,d1                   ;RAW wieder schliessen
        jsr     _LVOClose(a6)

* Zum Schluss immer die Lib schliessen!
        move.l  a6,a1                   ;DOS-Lib-Basis
        move.l  _SysBase,a6             ;Basis Exec
        jsr     _LVOCloseLibrary(a6)    ;Funktion "Schliessen"

fini    rts                             ;Return zum CLI

GetKey  move.l  d5,d1                   ;von RAW lesen
        move.l  a3,d2                   ;in diesen Puffer
        move.l  #1,d3                   ;1 Zeichen
        jsr     _LVORead(a6)            ;Lesen aufrufen
        rts

print   move.l  a0,a1                   ;kopiere Textzeiger
p1      addq.l  #1,a1                   ;+1
        cmp.b   #0,(a1)+                ;Null-Byte?
        bne     p1                      ;wenn nicht
        sub.l   a0,a1                   ;= Textlaenge

        move.l  a1,d3                   ;Laenge
        move.l  d5,d1                   ;Handle
        move.l  a0,d2                   ;Adresse Text
        jsr     _LVOWrite(a6)           ;Funktion "Schreiben"
        rts

* Datenbereich:

table   dc.l    Eingabe
        dc.l    Rechnen
        dc.l    Stoppen

keys    dc.b    'E','R','S'
count   equ     *-keys
        cnop    0,4
dosname dc.b    'dos.library',0
        cnop    0,2
name    dc.b    'RAW:40/100/580/80/Stop mit S',0
        cnop    0,2

E_text  dc.b    'Hier Eingabe',10,0
        cnop    0,2
R_text  dc.b    'Hier Rechnen',10,0
        cnop    0,2

buffer  ds.b    8
        cnop    0,4

hbuf    ds.b    10
```

Bild 5.6: Menü-Technik universell

Diesmal beginne ich am Anfang. Wir lesen wieder eine Taste, jetzt aber nur eine, weil wir keine Funktionstasten erwarten, bzw. diese ignorieren werden.

Force Uppercase

Nachdem das Zeichen in das Register d0 geladen wurde, soll es in einen Großbuchstaben gewandelt werden, falls es nicht schon ein Großbuchstabe ist. Force Uppercase nennen das die Fachleute in neudeutsch.

Wenn Sie einmal auf eine ASCII-Tabelle schauen, wird Ihnen sicherlich auffallen, daß sich die Klein- und die Großbuchstaben immer um 32 unterscheiden. 2 hoch 5 ist aber auch 32, sprich, im Fall von Kleinbuchstaben ist Bit 5 gesetzt. Folglich wird ein guter (sind Sie doch) Assembler-Programmierer nicht sagen »wenn der Code > Z ist, dann subtrahiere 32«, sondern er sagt einfach »lösche Bit 5«. Auch dafür kennt der 68000 einen Befehl, nämlich »BCLR« (Bit Clear).

```
bclr   #5,d0
```

löscht Bit 5 (setzt es auf 0) im Operanden (hier d0).

An dieser Stelle sind wir also soweit, daß wir prüfen können, ob der User E, R oder S eingegeben hat. Wenn Sie nun bitte einmal auf das Ende des Listings schauen: Da gibt es eine kleine Tabelle für diese drei Zeichen.

5.9 Location Counter und Equates

Darunter aber steht

```
count   equ   *-keys
```

Wow, das ist ein Ding! Fangen wir mit »equ« an. »equ« ist ein sogenanntes Equate (englisch), Sie lesen es aber am besten als »equal« = gleich).

Die Assembler-Direktive von zum Beispiel

```
Anton equ  4711
```

bedeutet, ab jetzt kann man anstatt 4711 auch Anton sagen. »JSR Anton« wäre dann ein gültiger Befehl. Prinzipiell ist das nichts weiter als reine Textverarbeitung. Der Assembler setzt einfach nachher für Anton immer eine 4711 ein. Wir müssen Anton aber jetzt auch als Konstante ansehen. Falls Sie Pascal können, betrachten Sie das »equ« wie

»const«, als C-Programmierer wie ein »#define«. Daher dürfen wir in diesem Programm auch beim Bezug auf »count« nicht »count« schreiben, »#count« ist Pflicht!

Das nächste »Wow« ist der »*«. Als erstes Zeichen in einer Programmzeile ist er ganz harmlos und bedeutet nur »Kommentar folgt«. Als Operand heißt er Location Counter (LC). Sie wissen, daß Befehle, je nach Anzahl und Art der Operanden verschieden viel Speicher belegen. Deshalb führt der Assembler einen Zähler, der sozusagen die bis zu jedem Befehl verbrauchten Bytes mitzählt. In diesem Sinn entspricht der LC dem PC (Programm Counter), mit dem nachher die CPU ein Programm verfolgt. Der Unterschied: Der LC wird auch durch Assembler-Direktiven inkrementiert, wie zum Beispiel »dc.b «, das ja auch Bytes (mit Daten) belegt.

In der Assembler-Syntax heißt nun aber der LC nicht LC, sondern »*«. Betrachten wir ein Beispiel. Im Listing hätte der LC zu Beginn der Zeile mit der Marke »keys« den Wert 100. Die Anweisung »dc.b E, R, S « läßt ihn auf 100 (für E), stellt ihn dann auf 101 (für R) und schließlich auf 102 (für S).

Die nächste Zeile mit der Marke »count« sieht den LC als 103. An dieser Stelle erfolgt aber ein Equate. Da in Equates auch Ausdrücke erlaubt sind, hat

```
count   equ   *-keys
```

die Wirkung von

```
count  =  LC - keys
```

Das in Zahlen ergibt

```
count  = 103-100
```

Damit hätten wir unsere symbolische Konstante 3. Warum habe ich da nicht einfach »count equ 3« geschrieben? Antwort: Das machen nur Anfänger!

Wenn wir nämlich später die Tabelle erweitern, können wir das tun, ohne die Zeile mit dem »equ« anfassen zu müssen. In jedem Assembler-Lauf (der muß dann eh sein) wird automatisch die richtige Anzahl eingesetzt. Außerdem: Tabellen können ganz schön lang sein, da kann man sich leicht verzählen.

5.10 Suchen mit DBcc

Nun müssen wir den Key (steht immer noch in D0) in der Tabelle »keys« suchen. Diesen Teil finden Sie unter »Suche Tasten-Code in Tabelle«. Das Problem: Auch »nicht gefunden« muß als gemeldet erkannt werden. Die Lösung ist eine DBcc-Schleife. Führen Sie sich bitte immer vor Augen:

```
DBcc    dn,loop
```

heißt: Verlasse die Schleife, wenn die Bedingung cc erfüllt ist, oder dn auf —1 gelaufen ist, sonst springe zu »loop«.

Vorab wird mit »lea keys,a0« ein Zeiger auf den Beginn der Tabelle gestellt. Der Trick: Der Schleifenzähler D1 wird mit Count geladen (hier 3), eine DBcc-Schleife läuft aber im Grenzfall bis —1, hier also über vier Schritte. Das heißt, wenn die Schleife wegen des »eq« (»equal« bedeutet gefunden) im »dbeq« verlassen wird, kann D1 nicht negativ sein. Ist es negativ, kommt das »bmi start« zur Wirkung (springe, wenn negativ).

Unter »Suche Adresse zu Key« taucht nun das nächste Problem auf. In der DBcc-Schleife lief D1 ja rückwärts. Es hat also diese Werte im Fall von »gefunden von«:

Key	D1
1	3
2	2
3	1

Ich brauche aber die Folge 0, 1, 2, um wieder auf die Adreßtabelle wie im vorigen Beispiel zugreifen zu können. Dieses ergäbe sich ganz einfach, wenn ich von Count D1 subtrahieren würde. (3—3=0, 3—2=1, 3—1=2). Leider erlaubt der 68000 den Befehl »sub d1,#count« nicht; wie soll er auch von einer Konstanten etwas subtrahieren?

Da hilft die Anweisung

```
neg d1
```

Ich negiere d1. War es 3, dann wird es —3. Darauf addiere ich Count, ergibt 0. Alte Regel: Wenn man nicht subtrahieren kann, muß man eben den negativen Wert addieren. Den Rest hatten wir schon. Als kleinen Unterschied habe ich hier nun anstatt »asl« »lsl« genommen (»logical shift left«). Der Unterschied: ASL schiebt arithmetisch korrekt, würde also auch das Vorzeichen berücksichtigen, wenn wir eines hätten. Damit wären wir wieder an der Stelle angekommen, die das vorige Programm ausmachte. Mit »lea table,a0« zeigen wir auf die Adreßtabelle, und den Rest kennen Sie schon, fast...

Ein Blick auf die Texte zeigt Ihnen, daß ich die Längenbytes weggelassen habe. Stattdessen sind alle Texte mit einem Null-Byte abgeschlossen. Diese Technik ist sehr praktisch, weil ich mich jetzt um die Textlänge (das Abzählen der Zeichen) überhaupt nicht mehr kümmern muß. Nun muß allerdings das Unterprogramm etwas mehr tun.

Hinzugekommen sind diese Zeilen:

```
print   move.l  aØ,a1               ;kopiere Textzeiger
p1      addq.l  #1,a1               ;+1
        cmp.b   #Ø,(a1)+            ;Null-Byte?
        bne     p1                  ;wenn nicht
        sub.l   aØ,a1               ;= Textlaenge
```

In der kleinen Schleife wird einfach das Null-Byte gesucht. Die Routine fängt nur einen Fall nicht ab, nämlich daß der String leer ist (nur aus einem Null-Byte besteht).

Wenn Sie nicht mit Unterprogrammen arbeiten, können Sie die Textlänge auch mittels des Location Counters zur Verfügung stellen. Das sähe dann so aus:

```
Text_1    dc.b 'Das ist ein Text'
Len_1     equ  *-Text_1
```

Die Ausgabe dieses Textes sähe dann zum Beispiel so aus:

```
        move.l  d5,d1               ;Handle
        move.l  #Text_1,d           ;Adresse Text
        moveq   #Len_1,d3           ;Länge
        jsr     _LVOWrite(a6)       ;Funktion "Schreiben"
```

Beachten Sie auch hier bitte: »moveq« erweitert automatisch auf ein Langwort, begrenzt aber die Textlänge auf 127 Zeichen. Darüber hinaus müßten Sie dann doch das etwas langsamere »move.l« nehmen.

Kapitel 6

Rationalisierung der Arbeit

Strukturierung von Assemblerprogrammen

Makros

Include Files

Module

Wissen Sie eigentlich, »wieviel Programm« ein professioneller Programmierer pro Tag schreibt? 6 (in Worten sechs) Zeilen pro Tag! Diese Zahl ergibt sich, wenn man den Gesamtaufwand an Tagen, beginnend bei der Problemanalyse über die ersten Vorentwürfe, das eigentliche Programmieren, das Testen, das Debuggen bis hin zur Dokumentation addiert und dann die Programmzeilen durch diese Anzahl Tage dividiert.

Dabei spielt die Programmiersprache keine Rolle. Es kommen immer diese sechs Zeilen dabei heraus.

Nun kann man natürlich mit sechs Zeilen Pascal viel mehr Wirkung erzielen, als mit sechs Zeilen Assembler. Um so wichtiger ist es, gerade in der Assembler-Programmierung alle Möglichkeiten zur Rationalisierung der Arbeit auszunutzen.

6.1 Strukturierung von Assembler-Programmen

Die wirkungsvollste Methode ist die Strukturierung der Programme. Damit ist primär nicht »WHILE...WEND« und Ähnliches gemeint (ist auch vorteilhaft, siehe 6.1.1), sondern die grundsätzliche Struktur des Programms. Im allgemeinen sieht doch ein Programm so aus:

- Logo
- Menü
- Warten auf Eingabe
- Reaktion auf die Eingabe

Mit Logo ist das Bild gemeint, mit dem sich das Programm beim Start meldet. Dann werden dem Bediener die möglichen Funktionen des Programms in einem (Haupt-) Menü angeboten und auf eine Eingabe gewartet. Die Eingabe wird interpretiert und daraufhin die entsprechende Funktion aufgerufen. Im Ansatz hatten wir diese Technik im Kapitel 5 schon praktisch geübt. In BASIC sähe das so aus:

```
100 PRINT "LOGO"
200 PRINT "MENU"
300 INPUT KEY
400 ON KEY GOSUB 1000,2000,3000,.....
410 GOTO LOGO
```

In Assembler ändert sich daran prinzipiell nichts. In der typischen Schreibweise dieser Sprache könnte man schreiben: (»bsr« heißt »Branch to Sub Routine«, also GOSUB)

```
                bsr  logo
loop            bsr  menu
                bsr  eingabe           ;Warte auf Eingabe
                bsr  rechne_adresse
                bsr  adresse           ;Funktion aufrufen
                bra  loop
;
;Beginn der Unterprogramme
```

Sie sehen, der wesentliche Ansatz ist die Gliederung in Unterprogramme. Diese Unterprogramme rufen selbst wieder Unterprogramme auf, und auch die »Unter-Unterprogramme« rufen gegebenenfalls nochmals Unterprogramme auf, und seien es nur Funktionen des Betriebssystems. Gerade letztere zeigen jedoch einige wichtige Eigenschaften, die Unterprogramme haben sollten, nämlich:

- universelle Verwendbarkeit und
- klar definierte, einheitliche Schnittstelle.

Ein gutes Beispiel sind die Routinen des oben gezeigten Rahmens. Es ist durchaus möglich, daß Sie zu einigen Punkten des Hauptmenüs Untermenüs anbieten müssen. Dann ist es doch sehr praktisch, wenn Sie dafür dieselben Unterprogramme wiederum benutzen können.

6.1.1 Struktur in der Sprache

Im Kapitel 4 hatte ich Ihnen ein Beispiel vorgestellt, das die Buchstaben von A bis Z druckt. Hier noch einmal der wesentliche Teil des Programms:

```
        lea     buffer,aØ
        move    #25,dØ
        move.b  #'A',d1

loop    move.b  d1,(aØ)+
        addq    #1,d1
        dbra    dØ,loop
```

Bild 6.1: Drucken von A bis Z diskret geschrieben

Was halten Sie nun von der folgenden Alternative? (Ich garantiere, wir sind immer noch in Assembler.)

```
for d1 = #'A' to #'Z' do
  move.b   d1,(aØ)+
endfor
```

oder:

```
move #'A',d2
while d2 le #'Z' do
  move.b  d2,(aØ)+
  addq    #1,d2
endwhile
```

Bild 6.2: Zwei Lösungen mittels Makros

Bild 6.2 ist nur ein kleiner Auszug aus der Makro-Sprache guter Assembler. Wenn ich Ihnen jetzt noch sage, daß der daraus entstehende Code der gleiche ist, wie der aus der diskreten Lösung von Bild 6.1, dann wird Sie das Thema sicherlich interessieren.

6.2 Makros

Makro ist die Kurzform von Makro-Befehl. Makro selbst heißt so viel wie groß. Prinzipiell ist ein Makro nur eine Zusammenfassung einer Gruppe von Einzelbefehlen, wie wir sie bisher kannten, unter einem neuen Namen. Man kann die Einzelbefehle auch Mikro-Befehle nennen. Leider ist die Makro-Sprache nicht genormt. Jeder Assembler hat da seine eigene Syntax, die des eben gezeigten GST-Assemblers ist sogar ziemlich ausgefallen. Ich bringe deshalb alle folgenden Beispiele in der Makro-Sprache der HiSoft-/Metacomco-Assembler (stimmen überein), die den »Makro-Dialekten« der meisten Assembler am ehesten entsprechen. Hier ein Beispiel:

```
CALLEXEC  macro
          move.l    _SysBase,a6
          jsr       _LVO\1(a6)
          endm
```

Dieser Makro realisiert die Funktion CALLEXEC (Funktion der Exec-Library aufrufen), wie wir sie schon kennen. Jeder Makro hat einen Namen, der im Label-Feld stehen muß, gefolgt von dem Schlüsselwort »macro«. Ein Makro endet mit dem Schlüsselwort »endm«. Zwischen »macro« und »endm« kann eine beliebige Anzahl von Befehlen stehen. Ist der Makro einmal definiert, kann er beliebig oft mit seinem Namen aufgerufen werden. Innerhalb eines Makros dürfen auch die Namen anderer,

dann schon vorher definierter Makros, stehen. Ob und wie tief Makros so geschachtelt werden dürfen, lesen Sie aber besser in Ihrem Handbuch nach.

Bild 6.3 bringt zwei Makros, so wie Sie sie einfach zu Beginn eines Programms tippen können.

```
CALLLIB MACRO
        JSR     \1(A6)
        ENDM

LINKLIB MACRO
        MOVE.L  \2,A6
        CALLLIB \1
        ENDM
```

Bild 6.3: Zwei Makros, die man immer braucht

Der Makro »CALLLIB« wie »Call Library« beinhaltet das Ihnen schon bekannte »jsr offset(a6)«. Wichtig ist hier, daß wir dem Makro einen Parameter übergeben müssen, nämlich den einzusetzenden String. Für solche Parameter haben Makros Variable. Beim GST-Assembler (demnächst verfügbar) dürfen dies Namen sein; meistens üblich sind aber Ziffern mit einem Schrägstrich oder (bei SEKA) einem Fragezeichen davor. Bei Metacomco sind auch noch die Buchstaben A bis Z erlaubt. Beachten Sie bitte, daß der Schrägstrich bei Metacomco und HiSoft ein »Backslash« (nach links gekippter Strich) sein muß. Solche Einschränkungen sind zwar unschön (das Zeichen gibt es nicht in jedem Editor) aber übliche Unsitte. Beim zweiten Makro haben Sie sicherlich schon erkannt, daß ein Makro einen anderen aufrufen darf. Dieser muß aber vorher definiert sein! Doch nun zur Praxis.

Das Programm soll lediglich das schon bekannte Hallo ausgeben. Bild 6.4 zeigt die Lösung. Vergleichen Sie diese bitte mit Bild 4.1. aus Kapitel 4.

```
        opt     l-                              ;nicht linken!

_SysBase          equ    4                      ;Basis von Exec
_LVOOpenLibrary   equ    -552                   ;Library oeffnen
_LVOCloseLibrary  equ    -414                   ;Library schliessen
_LVOOutput        equ    -60                    ;DOS: Output-Handle holen
_LVOWrite         equ    -48                    ;      Ausgabe

_main   move.l  #dosname,a1                     ;Name der DOS-Lib
        moveq   #0,d0                           ;Version egal
        LINKLIB OpenLibrary,_SysBase            ;DOS-Lib oeffnen
        tst.l   d0                              ;Fehler?
        beq     fini                            ;wenn Fehler, Ende
        move.l  d0,_DOSBase                     ;Zeiger merken

        LINKLIB Output,_DOSBase                 ;Hole Output-Handle

        print   d0,#string,20                   ;Text ausgeben

        move.l  _DOSBase,a1                     ;Basis der Lib
        LINKLIB CloseLibrary,_SysBase           ;Funktion "Schliessen"

fini    rts                                     ;Return zum CLI

_DOSBase dc.l   0
dosname  dc.b   'dos.library',0
         cnop   0,2
string   dc.b   'Hallo lieber Leser!',10
         cnop   0,2
```

Bild 6.4: Ein Programm mit Makros

Das sieht doch richtig gut aus. Was ist nun der Haken an der Sache? Es fehlen die Makros, die ich Ihnen mit Bild 6.5 vorstellen möchte.

```
LINKLIB MACRO
        IFNE    NARG-2
        FAIL    ---- Makro LINKLIB: Nicht 2 Argumente ----
        ENDC
        MOVE.L  A6,-(SP)
        MOVE.L  \2,A6
        JSR     _LVO\1(A6)
        MOVE.L  (SP)+,A6
        ENDM
```

```
print   MACRO
        IFNE     NARG-3
        FAIL     ---- Makro print: Nicht 3 Argumente ----
        ENDC
        MOVE.L   \1,D1                  ;Ausgabe-Handle
        MOVE.L   \2,D2                  ;Address Text
        MOVEQ    #\3,D3                 ;Laenge Text
        LINKLIB  Write,_DOSBase         ;Funktion "Schreiben"
        ENDM
```

Bild 6.5: Die Makros zu Bild 6.4

Den Makro LINKLIB finden Sie in ähnlicher Form unter vielen anderen in den Include-Files von Metacomco und HiSoft. Die Zeilen

```
IFNE     NARG-2
FAIL     ---- Makro LINKLIB: Nicht 2 Argumente ----
ENDC
```

können Sie prinzipiell auch weglassen. Da Sie aber in den Include-Files häufig anzutreffen sind (schauen Sie mal rein, es lohnt sich), sollte das aber doch erklärt werden. NARG ist eine Assembler-Variable und heißt »Number Arguments«. Diese Variable ist nur innerhalb eines Makros gültig (sonst Null) und hält die Anzahl der Parameter, mit denen der Makro aufgerufen wurde.

6.2.1 Bedingtes Assemblieren

Ob die Anzahl stimmt, ist eine andere Frage, aber das kann man prüfen. Dazu benutzt man eine zweite Eigenschaft guter Assembler, nämlich bedingtes Assemblieren. Hierfür gilt die generelle Form

```
IFcc
 tue das, wenn cc true
ENDC
 hier weiter, wenn cc false
```

Die Bedingungen »cc« sind prinzipiell die gleichen, wie wir sie schon von den Branch-Befehlen her kennen. Die Einschränkung ist allerdings, daß immer nur ein Argument gegen Null verglichen werden kann. Will ich also prüfen, ob die Anzahl der Makro-Argumente (NARG) stimmt, und im Fehlerfall eine Warnung ausgeben, so muß ich sagen:

wenn NARG minus 2 nicht gleich Null

oder `IFNE NARG-2`

Sie brauchen diese Tests in aller Regel nicht durchzuführen, der Assembler meckert dann nur etwas später. Habe ich nämlich zum Beispiel in einem Makro die Zeilen

```
print   macro
        MOVE.L    \1,D1           ;Ausgabe-Handle
        MOVE.L    \2,D2           ;Address Text
```

und rufen diesen Makro auf mit

```
        print d4
```

so entsteht daraus

```
        MOVE    d4,D1
        MOVE      ,D2
```

Bei der zweiten Zeile wird der Assembler mit Recht etwas monieren, der Fehler wird also erkannt. Die Lösung mit dem »IFcc« zeigt vielleicht die Herkunft des Fehlers besser.

Nun hätten wir noch ein kleines Problem. Nehmen wir an, Sie hätten folgenden Makro geschrieben:

```
        nonsens   macro
        loop      move     d1,d2
                  bra      loop
                  endm
```

Schreiben Sie nun in Ihrem Programm

```
        nonsens
        nonsens
```

so wird der Makroprozessor daraus folgende Zeilen entwickeln:

```
        loop      move     d1,d2
                  bra      loop
        loop      move     d1,d2
                  bra      loop
```

Spätestens beim zweiten »bra loop« wird der arme Assembler ins Schleudern geraten. Zu welchem »loop« soll er denn nun springen? Praktisch wird er mit einer Fehlermeldung aussteigen. Um so etwas zu vermeiden, gibt es nun in guten Assemblern immer eine Lösung. Die sinnvollste Art findet man zum Beispiel bei GST, wo man schreiben würde:

```
nonsens   macro
          LOCAL   loop
loop      move    d1,d2
          bra     loop
          endm
```

Damit wird »loop« zur lokalen, also nur innerhalb des Makros gültigen Variablen erklärt. In anderen Assemblern findet man die Form

```
$n anstatt »loop«
```

Für n ist eine Zahl zwischen 1-99 (oder 1-999) einzusetzen. Diese Zahl wird bei jedem Aufruf des Makros um eins hochgezählt. Wenn Sie mehrere Makros mit internen Labels verwenden, müssen Sie allerdings darauf achten, daß Sie mit unterschiedlichen und gegebenenfalls weit auseinanderliegenden Zahlen starten. Um noch einmal auf das Beispiel aus der Einleitung zurückzukommen, nämlich

```
for d2 = #'A' to #'Z' do
  .....
  .....

endfor
```

so ist des Rätsels Lösung ganz einfach. »for«, »=«, »to« und »endfor« sind Makros. »do« beispielsweise wird nur eine lokale Marke erzeugen, »for« wird d2 laden und »endfor« ein »dbra d2,marke« generieren. Eine andere Anwendung wäre diese:

```
ret       makro
          rts
          endm
```

In diesem Fall wird der Makroprozessor immer nur für jedes »ret«, das in einem Text auftaucht, ein »rts« einsetzen. Sie können auf diese Art jeden Assemblerbefehl neu definieren. Nun kommt es aber noch schlimmer, nämlich hiermit:

```
ret       makro
          dc.b    $C9
          endm
```

»ret« heißt Return in Z80-Assembler.

Für die Z80-Maschinensprache muß aber ein »ret« in »$C9« übersetzt werden. Sie können somit auf Ihrem Amiga in 68000-Assembler ein Programm in Z80-Assembler schreiben. So etwas nennt man Cross-Assembler. Möglich machen dies Makros. Nun wissen Sie auch, wie man einen neuen Computer in Assembler programmiert, wenn es für das gute Stück noch gar keinen Assembler gibt.

6.2.2 Nur Textverarbeitung

Bei Makros handelt es sich um eine reine Textverarbeitung, die mit Assembler an sich herzlich wenig zu tun hat. Bild 6.6 bringt einen kleinen Auszug aus einem Programm, das den Makro PRINT definiert und ihn dann zweimal aufruft.

```
PRINT   macro
        movem.l d0-d3/a6,-(sp)
        jsr     _LVOOutput(a6)
        move.l  d0,d1
        move.l  \1,d2
        move.l  \2,d3
        jsr     _LVOWrite(a6)
        movem.l (sp)+,d0-d3/a6
        endm

        PRINT   #msg1,#len1

        PRINT   #msg2,#len2
```

Bild 6.6: Ein Programmauszug mit Makros

Nun ist jeder gute Assembler in der Lage, ein sogenanntes Assembler-Listing zu erzeugen. Ein Beispiel dafür bringt Bild 6.7.

```
HiSoft GenAmiga Assembler 1.0  page 1

   4                          PRINT   macro
   5                          movem.l d0-d3/a6,-(sp)
   6                          jsr     _LVOOutput(a6)
   7                          move.l  d0,d1
   8                          move.l  \1,d2
   9                          move.l  \2,d3
  10                          movem.l (sp)+,d0-d3/a6
  12                          endm
  13
  14 0000001A +48E7F002        movem.l d0-d3/a6,-(sp)
  14 0000001E +4EAEFFC4        jsr     _LVOOutput(a6)
  14 00000022 +2200            move.l  d0,d1
  14 00000042 +243C00000072    move.l  #msg1,d2
  14 0000002A +263C0000000D    move.l  #len1,d3
  14 00000030 +4EAEFFD0        jsr     _LVOWrite(a6)
```

```
14 00000034 +4CDF400F          movem.l (sp)+,d0-d3/a6
15
16 00000038 +48E7F002          movem.l d0-d3/a6,-(sp)
16 0000003C +4EAEFFC4          jsr     _LVOOutput(a6)
16 00000040 +2200              move.l  d0,d1
16 00000042 +243C00000072      move.l  #msg2,d2
16 00000048 +263C00000012      move.l  #len2,d3
16 0000004E +4EAEFFD0          jsr     _LVOWrite(a6)
16 00000052 +4CDF400F          movem.l (sp)+,d0-d3/a6
17
```

Bild 6.7: Ein Assembler-Listing

Im ersten Feld nach den Zeilennummern stehen die Adressen. Der Assembler beginnt normalerweise bei Null. Mit ORG kann man eine absolute Startadresse vorgeben. Das Beispiel beginnt bei 1A, weil ich hier nur einen Auszug darstelle. Im zweiten Feld stehen die »Op-Codes«, sprich das, was der Assembler aus den einzelnen Befehlen macht. Dies ist die hexadezimale Darstellung der Maschinensprache, die der 68000 letztlich nur versteht. Vor den Op-Codes sehen Sie hier immer ein Plus-Zeichen. Das ist kein Vorzeichen, sondern nur ein Symbol, das anzeigen soll, daß diese Befehle aus einer Makro-Entwicklung stammen. Hiermit wären wir beim interessanten Teil angekommen. Sie sehen recht deutlich, daß sich die 16er- und die 15er-Zeilen wiederholen (bei Makros werden die Zeilen nicht hochgezählt). Es sind nur andere Werte (msg2 anstatt msg1, len2 anstatt len1) eingesetzt worden. Das heißt, mit Makros werden zwar die Quelltexte kürzer, der Objekt-Code hingegen wird um so länger, je häufiger ein Makro entwickelt wird. So ab dreifachem Aufruf lohnt es sich in der Regel, ein Unterprogramm einzusetzen, sofern dies sinnvoll ist. Ein gutes Antibeispiel ist das folgende in Bild 6.8.

```
FUNCDEF     MACRO
_LVO\1      EQU       FUNC_CNT
FUNC_CNT    SET       FUNC_CNT-6
            ENDM
FUNC_CNT    SET       4*-6

   FUNCDEF  Supervisor
   FUNCDEF  ExitIntr
   FUNCDEF  Schedule
   FUNCDEF  Reschedule
```

Bild 6.8: Der Makro FUNCDEF

Mit diesem Makro werden die LVOs bei Metacomco entwickelt. Bei HiSoft hat man diesen Umweg gespart und gleich die Offsets hingeschrieben. Damit Sie den Sinn dieses Makros verstehen, muß ich etwas vorwegnehmen: Die Library-Offsets sind immer negativ. Sie beginnen immer bei -30 und jeder Eintrag belegt 6 Byte (ob alle 6 genutzt werden, ist eine andere Frage (Antwort: nein). Der Makro addiert immer -6, deshalb wird bei -24 (4 * -6) begonnen. SET hat die gleiche Wirkung wie EQU, allerdings mit einem Unterschied: Mit SET kann einer Marke immer wieder ein neuer Wert zugewiesen werden. EQU ist wirklich eine Konstante im Sinn des Wortes. Den Rest dürften Sie verstehen, wenn ich Ihnen nun mit Bild 6.9 das Ergebnis von Bild 6.8 zeige.

```
FFFFFFE8    _LVOSupervisor    EQU    FUNC_CNT
FFFFFFE2    FUNC_CNT          SET    FUNC_CNT-6
FFFFFFE2    _LVOExitIntr      EQU    FUNC_CNT
FFFFFFDC    FUNC_CNT          SET    FUNC_CNT-6
FFFFFFDC    _LVOSchedule      EQU    FUNC_CNT
FFFFFFD6    FUNC_CNT          SET    FUNC_CNT-6
FFFFFFD6    _LVOReschedule    EQU    FUNC_CNT
FFFFFFDØ    FUNC_CNT          SET    FUNC_CNT-6
```

Bild 6.9: Das Ergebnis des Programms von Bild 6.8

Bei den Zahlen müssen Sie immer die Notation im 2er-Komplement sehen. Die führenden FFFFFF können Sie sich dann wegdenken. Das verbleibende E2 (hex) zum Beispiel ist dezimal 226. 226-256 ist dann -30. Beachten Sie, daß _LVOSupervisor zuerst mit EQU auf -24 (E8) gesetzt wird. Erst die folgende SET-Direktive ändert den Wert in -30 (E2).

6.3 Include-Files

Auch die Include-Anweisung kann man zuerst zusammen mit Makros sinnvoll einsetzen. Nehmen wir an, Sie haben die Makro-Definition von Bild 6.6 in einem Text-File mit dem Namen »Mac66« abgelegt. Dann können Sie das Programm von Bild 6.6 jetzt so schreiben, wie es Bild 6.10 zeigt.

```
        include "Mac66"

        PRINT  #msg1,#len1
        PRINT  #msg2,#len2

msg1    dc.b    'Hallo'
len1    equ     *-msg1
        ds.w    Ø
msg2    dc.b    'Amiga'
len2    equ     *-msg2
        end
```

Bild 6.10: Ein Programm mit Makros, die aus einem Include-File gelesen werden

Diese Methode ist sehr sinnvoll, denn Sie werden sicherlich in jedem Programm zahlreich DOS-Funktionen einsetzen. Wenn Sie sich diese Funktionen einmal als Makros definiert und in Ihrem »Mac-File« abgelegt haben, ersparen Sie sich nicht nur eine Menge an Tipperei, sondern auch einiges an Zeit für die Fehlersuche wegen nicht gemachter Tippfehler. Es macht durchaus nichts, wenn im jeweiligen Programm viele der Makros nicht genutzt werden. Sie erzeugen dann auch keinen Code. Wird die »Mac-Lib« (Kürzel für Library = Bibliothek) zu groß, kostet es natürlich Zeit, wenn Sie der Assembler bei jedem Lauf einlesen muß. Auf einer RAM-Disk kann dies unter Umständen zu Platzproblemen führen.

Deshalb sollten Sie Ihre »Lib« möglichst in mehrere kleine Files nach Sachgebieten aufteilen. Ob man die Sache allerdings so weit treiben muß, wie in den Include-Files von Metacomco und HiSoft, sei dahingestellt. Da gibt es zum Beispiel Makros, die die Library-Namen definieren. Auf diese Art muß man sich nicht diese Namen merken, dafür aber die der Makros. Besonders überflüssig ist der folgende Makro, oder will da etwa jemand die Exec-Lib öffnen?

```
EXECNAME        macro
        dc.b    'exec.library',Ø
        even
        endm
```

Dennoch ist es sehr empfehlenswert, die Include-Files der Assembler-Systeme zu nutzen. Schauen Sie sich diese Files auch ruhig einmal mit einem Editor an. Sie können daraus sehr viel über die Datenstrukturen des Amiga lernen (auf das Thema gehen wir später noch ausführlich ein).

6.4 Module

Module sind ein weiteres Hilfsmittel, die Programmierarbeit zu rationalisieren. Die dahinterstehende Philosophie ist, daß man ein großes Programm in viele einzelne, voneinander möglichst unabhängige Blöcke oder Abschnitte, kurz Module genannt, unterteilt. Eine Sprache wie Modula beispielsweise ist aus dieser Philosophie heraus geboren worden. In Assembler muß man zwei Arten von Modulen unterscheiden, nämlich
- Textmodule
- Code-Module

6.4.1 Textmodule

Textmodule hatten wir praktisch schon behandelt, es sind die Include-Files. Ein auf Textebene modulisiertes Programm, das den Arbeitstitel DED trägt, könnte beispielsweise so aussehen:

```
include    "dos.mac"
include    "bios.mac"
include    "ded_logo"
include    "ded_menu"
include    "ded_subs"
```

Vorteil der Textmodule ist zuerst, daß die Listings relativ kurz werden, wenn Sie immer einen Teil, der fertig geworden ist, als Textmodul ablegen. Wenn ich zum Beispiel bei der Entwicklung des Teils »ded_menu« bin, in dem natürlich ein Bug steckt, dann brauche ich mich nicht erst bis zur Zeile 477 vorzuarbeiten, sondern bin da schon bei Zeile 5.

Vorteil Nummer 2 wäre natürlich, daß man allgemein brauchbare Dinge, wie zum Beispiel die DOS-Lib immer wieder verwenden kann.

6.4.2 Code-Module

Ein Nachteil der Textmodule ist, daß sie bei jedem Lauf neu assembliert werden müssen. Die Abhilfe bringen die Code-Module. Dazu werden einzelne Blöcke getrennt assembliert und nachher vom Linker mit dem Hauptprogramm zusammengebunden. Das klingt sehr gut, bringt aber zuerst einiges an Mehrarbeit mit sich und stellt auch einige Anforderungen an den Linker und die gesamte Programmierumgebung an sich. Um es gleich zu sagen, der Aufwand lohnt sich nur bei größeren bis sehr großen Programmen.

Beginnen wir mit der Mehrarbeit im Programm. In einem kompletten Programm können Sie beispielsweise ohne weiteres sagen:

```
bsr print.
```

Steckt aber die Print-Routine in einem anderen Modul, so müssen Sie Ihrem Programm mitteilen, daß »print« eine externe Routine ist. Gleiches gilt für Variablennamen. Aus diesen Gründen muß der Assembler Direktiven der Art

```
          External,
          Global
und/oder  XREF
```

bieten. Diese Direktiven müssen Sie natürlich auch anwenden. Je nach Assembler ist die Sache mehr oder weniger weit getrieben, ist natürlich auch nicht genormt, sprich Sie müssen sich mit der Thematik auseinandersetzen und einiges an Lernpensum bewältigen.

Haben Sie sich dadurch nicht abschrecken lassen und Ihr Programm schön modulisiert, kommt das nächste Problem.

Vorausgesetzt, Ihr Linker kann beliebig viele Module binden (Vorsicht, einige erlauben nur Eingabezeilen von zum Beispiel 64 Zeichen!), dann ist das natürlich jedesmal eine irre Tipperei, wenn Sie den Linker aufrufen. Dazu sollte es nun eine von zwei Lösungen geben:

1. Der gesamte Lauf wird »im Batch« abgearbeitet (siehe Kapitel 4).
2. Der Linker bietet eine Anweisung wie »INPUT File-Name«. In diesem Fall schreiben Sie einmal alle Linker-Anweisungen in einen Text-File. Beim Aufruf des Linkers sagen Sie ihm dann, daß er diesen Text-File benutzen soll.

Alles in allem, die Sache ist doch recht umständlich. Ich kann Ihnen nur raten, stellen Sie das Thema Code-Module vorerst zurück. Erst wenn Sie den 68000-Assembler richtig beherrschen, und Sie sich an große Aufgaben wie beispielsweise die Entwicklung eines BASIC-Interpreters heranwagen, dann sollten Sie sich - nun allerdings dringend - wieder mit dem Thema Modulisierung auf Code-Ebene auseinandersetzen.

Andererseits ist modulare Programmierung in Assembler der einzige Weg, um auch bei mittelgroßen Programmen einigermaßen über die Runden zu kommen. Wie man dabei praktisch vorgeht, soll das folgende Beispiel zeigen.

Es soll ein Diskeditor entwickelt werden. Im Hauptmenü hat der Anwender die Auswahl unter den Kommandos »L)esen, E)ditieren, S(chreiben und Exit«. Wie das Menü angeboten wird, lasse ich erst einmal dahingestellt sein. Fest steht nur, daß die Buchstaben L, E, S und X eingegeben die entsprechende Aktion auslösen sollen. Sozusagen im Bestand habe ich einen Include-File, der eine Taste liest und daraufhin das zugeordnete Unterprogramm aufruft. Diesen File finden Sie als Auszug in Bild 6.11. Es ist ein Teil des »CASE X OF«-Programms aus Kapitel 5.

```
* start.icl

start
* lese Taste nach dØ.......
        bclr    #5,dØ           ;Erzwinge Grossbuchstaben
        lea     keys,aØ         ;Tabelle gueltige Keys
        move    #count,d1       ;deren Anzahl
search  cmp.b   (aØ)+,dØ        ;Key auf aktuellem Platz?
        dbeq    d1,search       ;wenn nicht, weitersuchen
                                ;bis Tabellenende
        tst     d1              ;Key gefunden?
        bmi     start           ;wenn nicht, auf ein Neues
        neg     d1              ;sub d1,#count
        add     #count,d1       ;ergibt Platznr. von Key
        lsl     #2,d1           ;die mal 4
        lea     table,aØ        ;Adr. der Routine
        move.l  Ø(aØ,d1.w),aØ   ;bestimmen
        jsr     (aØ)            ;und diese aufrufen
        bra     start           ;usw.
```

Bild 6.11: Startmodul als Include-File

Nachdem dieser Modul existiert, beginne ich nun mein neues Programm so, wie es Bild 6.12 zeigt.

```
        include  "start.icl"
Lese    rts
Edit    rts
Schreib rts

keys    dc.b    'L','E','S','X'
count   equ     *-keys
        ds.w    Ø
table   dc.l    Lese, Edit,Schreib,Exit
```

Bild 6.12: Ein neues Programm wird so begonnen

6.5 Top Down Bottom Up

Sie sehen sofort die Struktur des Programms. Was in den einzelnen Unterprogrammen passiert, interessiert zuerst gar nicht. Man kann nun hergehen und die einzelnen Unterprogramme nacheinander mit »Fleisch füllen«. Ist ein Unterprogramm fertig, wird es ausgetestet. Erst wenn es läuft, wird das nächste begonnen. Praktisch geht man sogar noch einen Schritt weiter. Zum Beispiel benötigt das Unterprogramm »Lese« eine Routine, die einen Sektor liest, und eine weitere, die den gelesenen Sektor auf dem Schirm in hex ausgibt. Dazu brauche ich unter anderem ein Unterprogramm »Anzeige«. Anzeige benötigt aber eine Routine, die ein Wort in die entsprechenden ASCII-Strings umwandelt. Damit ergibt sich dieser Ablauf:

```
Lese      bsr   read_sec
          bsr   anzeige
          rts

read_sec rts

anzeige   bsr  wandle
          bsr  print
          rts

wandle    rts
print     rts
```

Begonnen habe ich ganz oben und bin zum Schluß beim Unterprogramm »print« gelandet. Dies muß ich nun wirklich bearbeiten. Wenn die Routine »print« läuft, kann ich »wandle« beginnen. Denn nun erst kann ich ja die gewandelten Hex-Zahlen ausgeben und somit die Routine »print« auch testen. Jetzt werde ich mir »anzeige« vornehmen, das durch mehrfachen Aufruf von Print einen Pufferinhalt auf dem Schirm ausgibt. Danach werde ich »read_sec« schreiben, was diesen Puffer mit Daten füllt. Nun schließlich kann ich im Hauptmenü »Lese« aufrufen und wäre damit wieder ganz oben.

Dieses »von oben nach unten und wieder zurück« nennt man »top down bottom up«. Dies ist eine Methode der Programmierung, die gerade in Assembler sehr zu empfehlen ist. Sie beschränken damit die Fehlersuche immer nur auf einen kleinen, überschaubaren Bereich. Scheuen Sie dabei auch nicht den Mehraufwand, einzelne Unterprogramme temporär mit Spieldaten zu versorgen. Der Aufwand ist gering, der Nutzen ist groß. Zum Beispiel soll die Routine »wandle« ein Wort in D0 als Hex-String ausgeben. Schreiben Sie dann einfach

```
          move  $19AF,dØ
```

vorläufig als erste Zeile im Unterprogramm »wandle«. Wenn Sie nun die Routine testen und »19AF« auf dem Schirm sehen, dann können Sie schon ziemlich sicher sein, daß »wandle« funktioniert.

Kapitel 7

Programmentwicklung Schritt für Schritt

am Beispiel »bindec«

7.1 Das Prinzip der Konvertierung von Binärzahlen in Strings

In diesem Kapitel soll gezeigt werden, wie man ein Programm Schritt für Schritt entwickelt. Als nützliches Beispiel dient eine Routine, die wir später noch oft benötigen werden. Ihr Name ist »bindec«. »bindec« soll einen positiven Integerwert (0..65535), wie ihn der 68000 sieht, also binär, in einen Dezimalstring wandeln, den wir lesen können.

Bild 7.1 zeigt den ersten Schritt, an dem wir die grundsätzliche Technik der Zahlenwandlung studieren wollen.

Vorab: Es ist für den ersten Ansatz günstiger, jede Stelle (jedes Zeichen) sofort auszugeben und nicht erst alle Zeichen in einem Puffer zu sammeln. Außerdem gibt es viele Listings zu Computern, die zeichenorientiert arbeiten und dafür Routinen mit Namen wie CONOUT zur Verfügung stellen (Beispiel: Atari ST). Deshalb, und damit Sie auch »Fremdlistings« einfacher übernehmen können, simuliere ich ein CONOUT mittels des Makros gleichen Namens. CONOUT entspricht dem Makro PRINT aus Kapitel 6, nur daß hier die Länge konstant 1 ist und deshalb nicht mehr übergeben werden muß.

```
        opt      l-
* dec1
        include  OpenDos.i

CONOUT  macro
        movem.l d0-d3/a6,-(sp)
        jsr     _LVOOutput(a6)           ;Hole Output-Handle
        move.l  d0,d1                    ;Ausgabe-Handle
        move.l  \1,d2                    ;Address Text
        move.l  #1,d3                    ;Laenge Text
        jsr     _LVOWrite(a6)            ;Funktion "Schreiben"
        movem.l (sp)+,d0-d3/a6
        endm

        move     #62345,d2     ;Testzahl

        and.l    #$FFFF,d2     ;Begrenze auf Wort
        divs     #10000,d2     ;10000er Stelle
        bsr      out           ;ausgeben

        swap     d2            ;Divisionsrest nach d2.w
        and.l    #$FFFF,d2     ;wieder auf Wort bringen
        divs     #1000,d2      ;nun die 1000er Stelle
        bsr      out
```

```
        swap    d2              ;wie vor, die 100er
        and.l   #$FFFF,d2
        divs    #100,d2
        bsr     out

        swap    d2              ;nun die 10er
        and.l   #$FFFF,d2
        divs    #10,d2
        bsr     out

        swap    d2              ;und die 1er
        bsr     out

        move.b  #10,buffer
        CONOUT  #buffer

        move.l  a6,a1                       ;DOS-Lib-Basis
        move.l  _SysBase,a6                 ;Basis Exec
        jsr     _LVOCloseLibrary(a6)        ;Funktion "Schliessen"

fini    rts                                 ;Return zum CLI

out     add.b   #'0',d2                     ;in ASCII wandeln
        move.b  d2,buffer
        CONOUT  #buffer                     ;1 Zeichen ausgeben
        rts

* Datenbereich:

dosname dc.b    'dos.library',0
        cnop    0,2

buffer  ds.b    80
```

Bild 7.1: Das Prinzip von »bindec«

Das Prinzip der Zahlenwandlung ist ganz einfach. Wir müssen die Zahl, beispielsweise 123, ausdrücken als 1 Hunderter, 2 Zehner und 3 Einer. Wenn wir so die Ziffern 1, 2 und 3 isoliert haben, sind das »im Computer« zwar immer noch Zahlen, aber dann muß man darauf nur noch den ASCII-Code des Zeichens '0' addieren, und schon hat man druckbare Zeichen. Die Methode der Isolation der einzelnen Ziffern ist die fortlaufende Division und zwar so:

```
123 / 100  =  1  Rest  23
 23 /  10  =  2  Rest   3
  3 /   1  =  3  Rest   0
```

Das Dividieren wird beim 68000 mit Hilfe der Befehle

```
DIVS   oder DIVU
```

erledigt. Das heißt »Division Signed« (mit Vorzeichen) oder »Division Unsigned« (ohne Vorzeichen). Dividiert wird ein 32-Bit-Dividend durch einen 16-Bit-Divisor. Dividiert wird immer Ziel/Quelle. Danach steht das Ergebnis im Zieloperanden und zwar so:

höherwertiges Wort	niederwertiges Wort
Rest	Quotient

Das Programm von Bild 7.1 soll das Wort im Register D2 in »Dezi« wandeln. Weil wir nur Worte zulassen, muß ein eventueller Langwort-Dividend auf Wortlänge (in D2) mittels des »and.l«-Befehls begrenzt werden. Nun wird D2 durch 10 000 dividiert. Das Ergebnis ist der Wert der 10 000er-Stelle, der im Unterprogramm »out« ausgegeben wird. Nun wird mittels des SWAP-Befehls der Rest in das niederwertige Wort gebracht, dieser wird wieder »auf Wort« begrenzt und dann durch 1000 dividiert. Das setzt sich dann so mit der 100er-und der 10er-Stelle fort. Beim »Einer« müssen wir natürlich nicht mehr dividieren, nur vergessen dürfen wir ihn nicht.

So weit so gut. Nur wenn man sich das Programm so ansieht, fallen doch einige Wiederholungen auf. Da muß man doch rationalisieren können! Den ersten Ansatz dazu zeigt Bild 7.2.

```
        opt      l-
* dec2
        include  OpenDos.i
        include  conout.i

        move     #12345,d2      ;Testzahl

        move     #10000,d1      ;10000er Stelle
        bsr      out2           ;ausgeben

        move     #1000,d1       ;nun die 1000er Stelle
        bsr      out1

        move     #100,d1
        bsr      out1

        move     #10,d1
        bsr      out1
```

```
        move    #1,d1
        bsr     out1

        move.b  #1Ø,buffer
        CONOUT  #buffer

        move.l  a6,a1                   ;DOS-Lib-Basis
        move.l  _SysBase,a6             ;Basis Exec
        jsr     _LVOCloseLibrary(a6)    ;Funktion "Schliessen"

fini    rts                             ;Return zum CLI


out1    swap    d2                      ;Div.-Rest nach d2
out2    and.l   #$FFFF,d2               ;wieder auf Word
        divs    d1,d2                   ;Stelle holen
        add.b   #'Ø',d2                 ;in ASCII wandeln
        move.b  d2,buffer
        CONOUT  #buffer                 ;1 Zeichen ausgeben
        rts

* Datenbereich:

dosname dc.b    'dos.library',Ø
        cnop    Ø,2

buffer  ds.b    8Ø
```

Bild 7.2: Ratio-Schritt 1. Mehr Arbeit ins Unterprogramm

Wieder vorab: Der Makro CONOUT wurde inzwischen in den Include-File »conout.i« ausgelagert. Wählen Sie dazu einfach aus Bild 7.1 diesen Block im Editor aus und speichern ihn unter »conout.i« auf der Disk.

Sie sehen, die Befehle »swap«, »ext.l« und »divs« sind in das Unterprogramm gewandert. Da beim ersten Aufruf aber nicht »geswappt« werden darf, wurde das Unterprogramm mit zwei Einsprungstellen versehen. Das ist ein beliebter aber sehr unfeiner Trick. Der Divisor wird jeweils im Register D1 übergeben. Nun stört noch die Tatsache, daß da fünf nahezu gleiche Unterprogramm-Aufrufe existieren. Wie man das ändert, zeigt Bild 7.3.

```
        opt      l-
* dec3
        include  OpenDos.i
        include  conout.i

        move     #12345,d2     ;Testzahl

        move.l   #10000,d1     ;10000er Stelle. Nun Long!
        bsr      out2          ;ausgeben

        move     #3,d3
loop    divs     #10,d1
        bsr      out1
        dbra     d3,loop

        move.b   #10,buffer
        CONOUT   #buffer

        move.l   a6,a1                   ;DOS-Lib-Basis
        move.l   _SysBase,a6             ;Basis Exec
        jsr      _LVOCloseLibrary(a6)    ;Funktion "Schliessen"

fini    rts                              ;Return zum CLI

out1    swap     d2                      ;Div.-Rest nach d2.w
out2    and.l    #$FFFF,d2               ;wieder auf Word
        divs     d1,d2                   ;Stelle holen
        add.b    #'0',d2                 ;in ASCII wandeln
        move.b   d2,buffer
        CONOUT   #buffer                 ;1 Zeichen ausgeben
        rts

* Datenbereich:

dosname  dc.b    'dos.library',0
         cnop    0,2

buffer   ds.b    80
```

Bild 7.3: Ratioschritt 2: Eine Schleife hinzu

Die Schleife wurde nach altbekannter Art mittels »dbra« aufgebaut, wobei D3 als Zähler dient. Innerhalb der Schleife wird nun der Divisor D1 selbst immer durch 10 dividiert. Achten Sie bitte darauf, daß D1 nun mit einer langen Konstante initialisiert wird. Da die Schleife bis zum »Einer« laufen muß, wird dieser zum Schluß überflüssigerweise durch eins dividiert. Das abzufangen kostet aber mehr Zeit, also lassen wir das so

stehen. Nun meine ich aber, daß das Unterprogramm überflüssig ist. Wie man das UP in die Schleife bringt, zeigt Bild 7.4.

```
        opt      l-
* dec4
        include  OpenDos.i
        include  conout.i

        move     #123,d2                    ;Testzahl

        move.l   #10000,d1     ;10000er Stelle. Nun Long!
        move     #4,d3         ;Schleifenzaehler jetzt 4!
        bra      out2          ;ausgeben

loop    divs     #10,d1
out1    swap     d2                         ;Div.-Rest nach d2.w
out2    and.l    #$FFFF,d2                  ;wieder auf Word
        divs     d1,d2                      ;Stelle holen
        add.b    #'0',d2                    ;in ASCII wandeln
        move.b   d2,buffer
        CONOUT   #buffer                    ;1 Zeichen ausgeben
        dbra     d3,loop

        move.b   #10,buffer
        CONOUT   #buffer

        move.l   a6,a1                      ;DOS-Lib-Basis
        move.l   _SysBase,a6                ;Basis Exec
        jsr      _LVOCloseLibrary(a6)       ;Funktion "Schliessen"

fini    rts                                 ;Return zum CLI

* Datenbereich:

dosname  dc.b    'dos.library',0
         cnop    0,2

buffer   ds.b    80
```

Bild 7.4: Ratioschritt 3: Unterprogramme entfallen

```
        opt     l-
* dec5
        include OpenDos.i
        include conout.i

        move    #123,d2                         ;Testzahl

        clr     d4              ;Flag Nullunterdrueckung

        move.l  #10000,d1       ;10000er Stelle. Nun Long!
        move    #4,d3           ;Schleifenzaehler jetzt 4!
        bra     out2            ;ausgeben

loop    divs    #10,d1
out1    swap    d2              ;Div.-Rest nach d2.w
out2    and.l   #$FFFF,d2       ;wieder auf Word
        divs    d1,d2           ;Stelle holen
        add.b   #'0',d2         ;in ASCII wandeln

        cmp.b   #'0',d2         ;ist es eine Null?
        bne     out3            ;wenn nicht, ausgeben
        tst     d4              ;Blank erlaubt?
        bne     out3            ;nein: gebe Null aus
        move.b  #' ',d2         ;ja  : setze Blank ein
        bra     out4            ;      und raus damit
out3    move    #1,d4           ;Flag keine Blanks mehr

out4    move.b  d2,buffer
        CONOUT  #buffer         ;1 Zeichen ausgeben
        dbra    d3,loop

        move.b  #10,buffer
        CONOUT  #buffer

        move.l  a6,a1                   ;DOS-Lib-Basis
        move.l  _SysBase,a6             ;Basis Exec
        jsr     _LVOCloseLibrary(a6)    ;Funktion "Schliessen"

fini    rts                             ;Return zum CLI

* Datenbereich:

dosname  dc.b  'dos library',0
         cnop  0,2

buffer   ds.b  80
```

Bild 7.5: Schritt 4: Unterdrückung führender Nullen hinzu

Beachten Sie, daß ich wegen des beim ersten Mal unerwünschten »swap« jetzt in die Schleife hineinspringe, weshalb ich nun den Schleifenzähler mit 4 initialisieren muß. Jetzt aber genug der Ratio, kümmern wir uns lieber um die Schönheit. Ihnen ist sicherlich aufgefallen, daß das Programm führende Nullen schreibt, wenn die Zahlen kleiner als fünfstellig sind. Die unterdrückt man üblicherweise durch Ausgabe von Blanks anstatt führender Nullen. Wie man das macht, zeigt Bild 7.5.

Das Problem ist einfach zu beschreiben. Führende Nullen sollen durch Blanks ersetzt werden, andere Nullen natürlich nicht. Dazu benötigt man ein Flag (Merker), das gesetzt wird, sobald eine Zahl ungleich null auftaucht. Soweit die Logik. Praktischer ist jedoch, den Gedankengang etwas abzuwandeln, nämlich so: Jede Zahl ungleich null setzt das Flag. Damit kann man sich die sehr aufwendige Realisierung des Unterscheidens von erster »nicht Null« und anderen Nullen ersparen.

In unserem Fall ist das Register D4 das Flag. Die Abfrage beginnt in der Zeile mit »+++ neu +++« am rechten Rand. Ist die Zahl keine Null, wird D4 mit 1 geladen und dann das Zeichen ausgegeben. Ansonsten muß es eine Null sein. Und nun kommt der Test. Ist D4 gesetzt, wird die Null als Null ausgedruckt. Wenn nicht, wird D2 mit einem Blank geladen.

Nun habe ich beschlossen, das Ganze soll ein universell verwendbares Unterprogramm werden. Das heißt zuerst, das Unterprogramm darf die Zeichen nicht auf dem Schirm ausgeben, weil man sonst zum Beispiel nicht drucken kann. Die Änderung ist kein Problem. Bild 7.6 bringt die Lösung. Die Ausgabe erfolgt in einen Puffer namens »buffer«. Als Zeiger durch den Puffer wirkt das Register A0. Um zu wissen, wieviel Zeichen im Puffer gültig sind, wird als letztes Zeichen ein Null-Byte geschrieben. Das ist sehr praktisch, haben wir doch damit gleichzeitig einen DOS-String.

```
        opt     l-
* dec6
        include OpenDos.i
        include conout.i

PRINT   macro
        movem.l dØ-d3/a6,-(sp)
        jsr     _LVOOutput(a6)          ;Hole Output-Handle
        move.l  dØ,d1                   ;Ausgabe-Handle
        move.l  \1,d2                   ;Address Text
        move.l  \2,d3                   ;Laenge Text
        jsr     _LVOWrite(a6)           ;Funktion "Schreiben"
        movem.l (sp)+,dØ-d3/a6
        endm

        move    #123,d2                         ;Testzahl
```

```
        clr     d4              ;Flag Nullunterdrueckung
        lea     buffer,a0       ;Ergebnis-Puffer

        move.l  #10000,d1       ;10000er Stelle. Nun Long!
        move    #4,d3           ;Schleifenzaehler jetzt 4!
        bra     out2            ;ausgeben

loop    divs    #10,d1
out1    swap    d2                          ;Div.-Rest nach d2.w
out2    and.l   #$FFFF,d2                   ;wieder auf Word
        divs    d1,d2                       ;Stelle holen
        add.b   #'0',d2                     ;in ASCII wandeln

        cmp.b   #'0',d2                     ;ist es eine Null?
        bne     out3                        ;wenn nicht, ausgeben
        tst     d4                          ;Blank erlaubt?
        bne     out                         ;nein: gebe Null aus
        move.b  #' ',d2                     ;ja  : setze Blank ein
        bra     out4                        ;      und raus damit
out3    move    #1,d4                       ;Flag keine Blanks mehr
out4    move.b  d2,(a0)+                    ;Zeichen -> Puffer
        dbra    d3,loop
        move.b  #0,(a0)                     ;von PRINT ignoriert
        PRINT   #buffer,#5                  ;Zahl ausgeben

        move.w  #$0A0A,buffer               ;2 Linefeeds
        PRINT   #buffer,#2

        move.l  a6,a1                       ;DOS-Lib-Basis
        move.l  _SysBase,a6                 ;Basis Exec
        jsr     _LVOCloseLibrary(a6)        ;Funktion "Schliessen"

fini    rts                                 ;Return zum CLI

* Datenbereich:

dosname  dc.b   'dos.library',0
         cnop   0,2

buffer   ds.b   80
```

Bild 7.6: Schritt 5: Ausgabe in einen String

Nur ein richtig universelles Unterprogramm ist es damit trotzdem noch nicht. Störend ist schon, daß die Wertübergabe im Register D2 erfolgt, das könnte ja das Hauptprogramm benutzen. Schlimm jedoch ist, daß eine Variable mit dem Namen »buffer« fest mit diesem Programm gekoppelt ist. Das muß geändert werden. Wie, zeigt Bild 7.7.

```
        opt      1-
* dec7
        include  OpenDos.i
        include  conout.i
PRINT   macro
        movem.l d0-d3/a6,-(sp)
        jsr     _LVOOutput(a6)          ;Hole Output-Handle
        move.l  d0,d1                   ;Ausgabe-Handle
        move.l  \1,d2                   ;Address Text
        move.l  \2,d3                   ;Laenge Text
        jsr     _LVOWrite(a6)           ;Funktion "Schreiben"
        movem.l (sp)+,d0-d3/a6
        endm

        move     #1001,-(sp)    ;Testzahl
        pea      buffer         ;Ergebnis-Puffer
        bsr      bindec         ;Routine aufrufen

        PRINT    #buffer,#5             ;Zahl ausgeben

        move.w   #$0A0A,buffer          ;2 Linefeeds
        PRINT    #buffer,#2

        move.l  a6,a1                   ;DOS-Lib-Basis
        move.l  _SysBase,a6             ;Basis Exec
        jsr     _LVOCloseLibrary(a6)    ;Funktion "Schliessen"

fini    rts                             ;Return zum CLI

;-----------------------------------------------------------------
bindec  move.l   4(sp),a0       ;Pufferadresse holen
        move     8(sp),d2       ;und die zu wandelnde Zahl
        move.l   #10000,d1      ;erster Divisor
        move     #4,d3          ;Schleifenzaehler
        clr      d4             ;Flag Nullunterdrueckung
        lea      buffer,a0      ;Ergebnispuffer
        bra      out2           ;10000er wandeln
loop    divs     #10,d1         ;und nun die 1000er bis 1er
out1    swap     d2             ;Divisionsrest nach d2.w
out2    and.l   #$FFFF,d2       ;wieder auf Word
        divs     d1,d2          ;naechste Stelle
        add.b    #'0',d2        ;wandle in ASCII

        cmpi.b   #'0',d2        ;ist es eine Null?
        bne      out3           ;wenn nicht ausgeben
        tst      d4             ;Blank erlaubt?
        bne      out3           ;nein, gebe die Null aus
        move     #' ',d2        ;setze Blank ein
        bra      out4           ;und gebe aus
out3    move     #1,d4          ;Flag keine Blanks mehr
```

```
out4    move.b  d2,(aØ)+        ;Zeichen -> Puffer
        dbra    d3,loop
        move.b  #Ø,(aØ)         ;Abschlusszeichen

        move.l  (sp)+,aØ        ;hole Return-Adresse
        addq.l  #6,sp           ;Parameter vom Stack
        jmp     (aØ)            ;und Return


* Datenbereich:

dosname   dc.b   'dos.library',Ø
          cnop   Ø,2

buffer   ds.b   8Ø
```

Bild 7.7: Schritt 6: Parameterübergabe über den Stack

Aufgerufen wird das Unterprogramm »decbin« in der Folge
- Wert auf den Stack
- Pufferadresse auf den Stack
- `bsr decbin`

Danach befinden sich auf dem Stack bei
8(sp) : Wert
4(sp) : Pufferadresse
0(sp) : Return-Adresse

Folglich kann man sich mit

```
        move.l  4(sp),aØ
        move.w  8(sp),d2
```

diese Parameter leicht holen. Der Rest läuft dann wie bekannt, allerdings mit einem kleinen Unterschied zum Schluß.

```
        move.l  (sp)+,aØ
```

holt die Return-Adresse vom Stack und inkrementiert den Stackpointer um 4. Nun muß ich aber noch die 6 Byte der Parameter (Wort für Wert und Langwort für Adresse) vom Stack »entfernen«. Das geschieht mittels der Anweisung

```
        addq.l  #6,sp
```

Zum Schluß muß man natürlich »returnen«, was aber jetzt nur noch heißt, »springe zur Return-Adresse«, also

```
        jmp     (aØ)
```

Der Mechanismus kommt Ihnen bekannt vor? Sie haben recht, so ähnlich arbeiten Hochsprachen. Das ist ganz interessant zu wissen: Die Sprache C, die die ideale Sprache für den Amiga sein will, denkt leider völlig an dessen Parametertransfer vorbei, der bekanntlich über Register läuft. Folglich packt jede C-Funktion immer brav alle Parameter auf den Stack. Dann fügt der Compiler eine Routine ein, die die Parameter wieder vom Stack holt und in die Register umlädt. Daß dieser Umweg ganz schön Zeit und Code kostet, können Sie sich wohl vorstellen. Dennoch sollten Sie den Mechanismus gut studieren, denn den müssen Sie kennen, wenn es, wie im entsprechenden Kapitel gezeigt, um die Einbindung von Assembler-Routinen in BASIC geht.

```
bindec  movem.l  d1-d4,-(sp)    ;Register retten
        move.l   2Ø(sp),aØ      ;Pufferadresse holen
        move     24(sp),d2      ;und die zu wandelnde Zahl
        move.l   #1ØØØØ,d1      ;erster Divisor
        move     #4,d3          ;Schleifenzaehler
        clr      d4             ;Flag Nullunterdrueckung
        lea      buffer,aØ      ;Ergebnispuffer
        bra      bd3            ;1ØØØØer wandeln
bd1     divs     #1Ø,d1         ;und nun die 1ØØØer bis 1er
bd2     swap     d2             ;Divisionsrest nach d2.w
bd3     and.l    #$FFFF,d2      ;wieder auf Word
        divs     d1,d2          ;naechste Stelle
        add.b    #'Ø',d2        ;wandle in ASCII

        cmpi.b   #'Ø',d2        ;ist es eine Null?
        bne      bd4            ;wenn nicht ausgeben
        tst      d4             ;Blank erlaubt?
        bne      bd4            ;nein, gebe die Null aus
        move     #' ',d2        ;setze Blank ein
        bra      bd5            ;und gebe aus
bd4     move     #1,d4          ;Flag keine Blanks mehr

bd5     move.b   d2,(aØ)+       ;Zeichen -> Puffer
        dbra     d3,bd1
        move.b   #Ø,(aØ)        ;Abschlusszeichen
        movem.l  (sp)+,d1-d4    ;Register zurueck
        move.l   (sp)+,aØ       ;hole Return-Adresse
        addq.l   #6,sp          ;Parameter vom Stack
        jmp      (aØ)           ;und Return
```

Bild 7.8: Der letzte Schritt. Arbeitsregister werden gesichert

So weit so gut, aber perfekt sind wir immer noch nicht. Unser »bindec« zerstört leider die Register D1 bis D4. A0 wird zwar auch geändert, aber das ist üblich, auch D0 ist immer »scratch« (Schmierpapier). Im letzten Schritt, dem im Bild 7.8, soll das auch abgestellt werden.

Vorab, ich habe die Labels so geändert, daß Sie hoffentlich in anderen Programmen nicht vorkommen. Falls Ihr Assembler lokale Labels bietet, dann sollten Sie davon Gebrauch machen (bei Metacomco $n...). Neu ist der Befehl

```
movem.l  d1-d4,-(sp)
```

Damit kann eine ganze Gruppe oder Liste von Registern auf den Stack gebracht werden. Auch Schreibweisen wie

```
movem.l  d1-d4/a1-a2/a5,-(sp)
```

sind zugelassen. Das Gegenstück (vom Stack holen) lautet dann

```
movem.l  (sp)+,d1-d4/a1-a2/a5
```

Da sich in unserem Beispiel nun vier Register, also 16 Byte auf dem Stack zusätzlich befinden, müssen wir nun, um die Parameter zu holen, auch diese 16 Byte zugeben, daher:

```
move.l   20(sp),a0      ;Pufferadresse holen
move     24(sp),d2      ;und die zu wandelnde Zahl
```

Nun speichern Sie bitte die Routine »bindec« in einem Extra-File, wir werden sie später noch benötigen.

Kapitel 8

Ein Schnellkurs in Sachen Intuition

8.1 Multitasking

Vorab: Läßt man ein eigenes Programm unter CLI laufen, wird das wie ein Unterprogramm des CLI gehandhabt. Es kann daher ohne Startup-Code geschrieben werden und schlicht mit RTS enden. Soll ein Programm hingegen als eigener Task laufen, muß es mit einem Startup-Code versehen werden (kommt in Kapitel 9). Der Sinn des Startup-Codes ist unter anderem, daß ein Programm zuerst auf eine Message (Nachricht) warten muß, bevor es loslegen darf, und damit wären wir beim Thema Multitasking.

Prinzipiell kann Ihr Task (Programm) so tun, als sei er der einzige Task im System. Praktisch kommen Sie damit nicht weit, denn Sie müssen mit anderen Tasks kommunizieren, um zum Beispiel zu erfahren, ob die Maus bewegt wurde. Die Kommunikation erfolgt beim Amiga über Message-Ports. Gesteuert wird das Ganze vom Message-Dispatcher, den Sie sich wie das Mädchen in der Telefonzentrale vorstellen können, das die Verbindungen zwischen den einzelnen Teilnehmern herstellt. Nur haben wir eine Super-Telefonistin. Sind Sie nämlich gerade besetzt, schreibt das Mädchen alles auf und gibt Ihnen die Nachricht durch, sobald Sie wieder frei sind. Etwas fachlicher: Die Nachrichten werden in einem Message-Queue gepuffert. Sie müssen natürlich mindestens ein Telefon haben, sprich mindestens einen Message-Port für Ihren Task einrichten.

Typisch für einen Task ist, daß er auf eine Nachricht wartet, sprich der User endlich mal eine Taste drückt oder die Maus bewegt. Dahinter steckt: Alle Eingaben laufen über einen Task mit dem Namen »input.device«. Der Amiga sorgt schon dafür, daß die Nachricht zu Ihrem Task (Ihrem Fenster) gelangt, nur Sie müssen darauf warten. Die unfeinste Methode ist nun das sogenannte Polling. Dazu geht der Task einfach in eine Schleife, in der er den Message-Port so lange abfragt, bis dort eine Nachricht anliegt. Damit verbraucht er nur Rechnerzeit, die anderen Tasks dann fehlt. Besser ist es, eine Funktion mit dem Namen Wait zu benutzen. In diesem Fall wird der Task aus der Liste der aktiven Tasks gestrichen und geht auf die Liste der wartenden Tasks, wo er den Betrieb nicht mehr aufhält. Man sagt auch, der Task schläft. Wach bleibt aber das Betriebssystem, das ständig prüft, ob eine Nachricht für den Task kommt. Ist das der Fall, wird der Task wieder geweckt (er setzt nach dem Wait-Aufruf fort) und kann nun seine Nachricht behandeln, zum Beispiel auf einen Tastendruck so reagieren, wie es das Programm vorsieht.

Die einfachste Form von Wait ist die Funktion WaitPort( ). Damit wartet man auf einen Port. Nun kann der Task aber mehrere davon haben, nicht aber auf alle Ports warten wollen. Für diesen Zweck gibt es die Signal-Bits. Jeder Port wird einem von 32 Bit zugeordnet, jeder Task hat seine eigenen Signal-Bits. Ein Task kann allerdings nur maximal 16 Bit belegen, die anderen 16 braucht das System. Um nun auf eine beliebige Port-Kombination zu warten, muß man nur die entsprechenden Bits »verodern« (ihre Werte addieren) und damit Wait( ) aufrufen (im Gegensatz zu WaitPort). Ein

Task sendet eine Message mit PutMsg( ). Erreicht diese Nachricht einen schlafenden Task (einen der Wait( ) aufgerufen hatte), wird er damit geweckt. Der Empfänger-Task wird nun mit GetMsg( ) die Nachricht lesen. Er sollte dann mit ReplyMsg( ) den Empfang quittieren. Letzteres ist bei einigen Tasks, zum Beispiel Intuition, Pflicht.

Das Thema Multitasking ist hiermit noch bei weitem nicht vollständig behandelt, doch wir müssen uns jetzt erst einmal um einige andere Amiga-Spezialitäten kümmern, um danach wieder automatisch beim Multitasking zu landen.

8.2 Screens, Windows und Gadgets

Neben den Libraries gibt es noch drei grundlegende Dinge, die man kennen sollte, nämlich Screens, Windows und Gadgets. Der Amiga erlaubt, mehrere virtuelle Bildschirme einzusetzen. Auf jedem Screen können sich wiederum mehrere Windows befinden. Der äußerliche Unterschied: Ein Screen kann immer nur vertikal verschoben werden, seine vorgewählte Größe kann nicht geändert werden. Einen Screen zu schaffen, ist recht einfach. Man definiert eine Struktur, in der die gewünschten Parameter eingetragen werden. Dann ruft man OpenScreen( ) auf. Die Funktion gibt einen Zeiger zurück oder NULL, wenn etwas schiefging. Dieser Zeiger muß dann als ein Parameter in die Window-Struktur eingetragen werden. Die wesentlichen Screen-Parameter sind die Auflösung (der Amiga kennt derer vier) und die Tiefe, womit die Anzahl der Bit-Planes gemeint ist. Mit einer Tiefe von zum Beispiel 3 sind 2^3 = 8 Farben möglich.

Ein Window wird sinngemäß wie ein Screen geöffnet, nur daß jetzt hierfür eine Window-Struktur definiert werden muß. Generell ist das Window das Element, das am mächtigsten ist, und mit dem Sie wohl auch am meisten umgehen werden. Es ist zum Beispiel nicht unbedingt nötig, einen Screen zu öffnen. Trägt man nämlich in der Window-Struktur für den Parameter Screen NULL ein (und als Typ WBENCHSCREEN), wird automatisch der Workbench-Screen benutzt. Die Struktur selbst ist nun folgende:

LeftEdge, TopEdge:	Linke, obere Ecke. 0,0 wäre links oben
Width, Height:	Breite und Höhe, mit denen das Fenster geöffnet werden soll
DetailPen:	Farbregister, mit denen die Details (zum Beispiel Gadgets) gezeichnet werden sollen (normalerweise 0).
BlockPen:	Farbregister zum Füllen von Flächen, normalerweise 1
IDCMPFlags:	siehe unten

Flags: siehe unten

FirstGadget: Zeiger auf das erste User-Gadget (NULL wenn keine)

CheckMark: Zeiger auf die Check-Marke für Menüs oder NULL, wenn die System-Marke benutzt werden soll.

Title: Zeiger auf einen Text (Fenster-Titel)

Sreen: Der Zeiger von OpenScreen, zu dem das Window gehören soll.

BitMap: Zeiger auf eine Super-Bitmap (meistens NULL)

MinWidth: Minimale Breite, auf die das Fenster verkleinert werden darf.

MinHeight: Minimale Höhe

MaxWidth: Maximale Breite

MaxHeight: Maximale Höhe

Type: Es gibt die Typen WBENCHSCREEN und CUSTOMSCREEN. Letzteren setzt man ein, wenn man einen eigenen Screen benutzt, sonst gilt der Workbench-Screen.

Beachtenswert ist, daß man nach dem Öffnen diese Struktur nicht mehr braucht. Üblicherweise wird man sie aber modifizieren und damit ein anderes Fenster öffnen. Natürlich hat niemand Lust, in jedem Programm diese lange Struktur einzutippen. Sie wird deshalb einfach als Include-File abgelegt und nachgeladen. Dann hat man noch eine kleine Funktion, die die paar Parameter, die wirklich geändert werden müssen, einträgt. Womit eines klar sein dürfte: Mittels der Angaben in der Open-Struktur generiert Intuition seine eigene Struktur, auf die dann der OpenWindow( ) zurückgegebene Zeiger zeigt. Diesen Zeiger sollte man gut aufheben, er wird noch des öfteren gebraucht. Nun aber zu den Dingen, die ich ausgelassen habe:

IDCMP heißt Intuition Direct Communication Message Port. Damit wären wir wieder beim Multitasking. Intuition stellt uns aber ein sehr komfortables User-Interface zur Verfügung. Dieses sorgt unter anderem dafür, daß zwei Message-Ports (für jede Richtung einer) geöffnet werden, sofern wir beim Öffnen IDCMP-Flags setzen. Diese Flags sind Bits, mehrere Flags kann man setzen, indem man die Bits »verodert«.

Die Kommunikation zwischen Intuition und einem Window läuft nun im wesentlichen über sogenannte Gadgets. Das sind Bedienelemente. Typisch für ein Window sind zum Beispiel das Close-Gadget (links oben) oder das Size-Gadget (rechts unten), womit Sie ein Window schließen bzw. in der Größe verändern können. Es handelt sich hier um System-Gadgets (im Gegensatz zu den User-Gadgets).

Beachten Sie bitte einen ganz wesentlichen Unterschied zum GEM des Atari ST. Bei den hier genannten System-Gadgets kümmert sich Intuition automatisch um die Aktionen, die der User mit den Gadgets anstellt, es sei denn, Sie verbieten das extra. Der User kann also ein Fenster verschieben oder seine Größe ändern, ohne daß Sie das in Ihrem Programm behandeln müssen. Im GEM ist es so, daß das AES (entspricht Intuition) dem Programm nur meldet, daß diese Aktion stattfindet. Das Programm muß dann selbst in einer Schleife ständig diese Meldungen abfragen und gegebenenfalls die entsprechenden Funktionen aufrufen. Wir sind hier also ein ganz riesiges Stück fortschrittlicher als im GEM, müssen aber auch wissen, wie man mit diesem komfortablen Instrument umgeht. Dazu mit Bild 8.1 wieder ein Beispiel.

```
* window1

* In diesen Files stecken diverse Deklarationen und Makros.
* Schauen Sie mal rein!

        incdir  ":include/"

        include intuition/intuition.i
        include intuition/intuition_lib.i
        include exec/exec_lib.i
        include graphics/graphics_lib.i

* Intuition Library oeffnen:
* --------------------------
        lea     intname,a1
        moveq   #Ø,dØ
        CALLEXEC OpenLibrary
        tst.l   dØ
        beq     abbruch
        move.l  dØ,_IntuitionBase    ;Basis-Zeiger sichern

* Graphics Library oeffnen
* ------------------------
        lea     grafname,a1
        moveq   #Ø,dØ
        CALLEXEC OpenLibrary
        tst.l   dØ
        beq     closeint
        move.l  dØ,_GfxBase          ;Basis-Zeiger sichern
```

```
* Window oeffnen
* --------------
        lea     windowdef,aØ          ;zeige auf Window-Struktur
        CALLINT OpenWindow            ;oeffne Window
        tst.l   dØ                    ;ging was schief?
        beq     closegraf             ;wenn ja
        move.l  dØ,windowptr          ;Window-Zeiger sichern

* Text im Fenster zeichnen
* ------------------------
        moveq   #1ØØ,dØ               ;X-Position
        moveq   #5Ø,d1                ;Y
        move.l  windowptr,a1          ;Via Window-Zeiger
        move.l  wd_RPort(a1),a1       ;  Rast-Port-Adresse holen
        CALLGRAF Move                 ;Funtion Move to X,Y

        move.l  windowptr,a1          ;brauche wieder Rastport
        move.l  wd_RPort(a1),a1
        lea     msg,aØ                ;Adresse Text
        moveq   #msglen,dØ            ;seine Laenge
        CALLGRAF Text                 ;und ausgeben

* Auf Event warten (kann hier nur WINDOWCLOSE sein)
* -------------------------------------------------
        move.l  windowptr,aØ          ;zeige auf Window-Struktur
        move.l  wd_UserPort(aØ),aØ    ;nun auf Message-Port
        move.b  MP_SIGBIT(aØ),d1      ;Anzahl Signal Bits -> d1
        moveq.l #1,dØ                 ;in Maske
        lsl.l   d1,dØ                 ;       konvertieren
        CALLEXEC Wait                 ;Schlaf gut!

* Fenster schliessen
* ------------------
        move.l  windowptr,aØ          ;wir sind wieder wach
        CALLINT CloseWindow           ;Fenster zu

* Libraries schliessen
* --------------------
closegraf
        move.l  _GfxBase,a1
        CALLEXEC CloseLibrary

closeint
        move.l  _IntuitionBase,a1
        CALLEXEC CloseLibrary

abbruch
        move.l  #Ø,dØ                 ;oder normales Ende
        rts

W_Gadgets equ   WINDOWSIZING!WINDOWDRAG!WINDOWDEPTH!WINDOWCLOSE
W_Extras  equ   SMART_REFRESH!ACTIVATE
```

```
W_Title dc.b    'Fenster-Titel',0

windowdef
        dc.w    200,50              ;links, oben
        dc.w    300,100             ;Breite, Hoehe
        dc.b    -1,-1               ;Pens des Screen
        dc.l    CLOSEWINDOW         ;einziges IDCMP Flag
        dc.l    W_Gadgets!W_Extras  ;Window Flags
        dc.l    0                   ;keine User-Gadgets
        dc.l    0                   ;keine User-Checkmark
        dc.l    W_Title             ;Titel des Window
        dc.l    0                   ;kein eigener Screen
        dc.l    0                   ;keine Super Bitmap
        dc.w    100,20              ;Min. Groesse
        dc.w    640,200             ;Max.
        dc.w    WBENCHSCREEN        ;Use Workbench Screen

intname         INTNAME             ;Name Intuition Lib (via Makro)
grafname        GRAFNAME            ;Name Graphics Lib


msg             dc.b    'Hello, World! '
msglen          equ     *-msg

_IntuitionBase  ds.l    1           ;Speicher fuer Zeiger
_GfxBase        ds.l    1
windowptr       ds.l    1
```

Bild 8.1: Ein erstes Intuition-Window

Das Programm soll ein Window auf den Schirm bringen und in das Window einen Text schreiben (zeichnen). Das Fenster soll auf dem Schirm verschiebbar sein, seine Größe darf geändert werden. Das Programm soll enden, wenn die Close-Box des Fensters angeklickt wird.

Dazu öffne ich zwei Libraries, nämlich Intuition und Graphics. Generell braucht man immer beide, ihre Basis-Zeiger sollte man sofort sichern, die werden häufig benötigt. Die Exec-Library ist sowieso immer dabei. Das obligatorische

```
        move.l  SysBase,a6
        jsr     _LVOxxx(a6)
```

ist im Makro

```
        CALLEXEC    xxx
```

versteckt. Sinngemäß funktioniert der Makro CALLINT, der eine Funktion der Intuition-Library aufruft. Intuition ist für die komplexen Dinge wie Fenster zuständig, Graphics hingegen für die Grundroutinen wie Zeichnen von Linien, Flächen oder Texten. Die beiden Instanzen entsprechen dem AES bzw. VDI des GEM des Atari ST.

Um ein Window zu öffnen, reicht der einfache Aufruf, wie im Listing gezeigt. Alles weitere steht in der Struktur ab Label »windowdef«. Als einziges IDCMP-Flag habe ich CLOSEWINDOW angegeben. Das heißt, nur wenn der User das Fenster schließt, meldet mir das Intuition. Alle anderen Events (Ereignisse) behandelt Intuition selbst. Natürlich kann man noch weitere Events zulassen, zum Beispiel MOUSEBUTTONS oder MOUSEMOVE, um nur einige zu nennen.

In der Folgezeile steht, was Intuition zu bearbeiten hat, nämlich Window_Gadgets und Extras. Ich habe das ein paar Zeilen höher als Equates hingeschrieben. Beide Equates hätten auch im »dc.l« stehen können, nur wäre dann das Listing zu breit geworden. In der ersten Equate-Zeile stehen die Window-Gadgets, die installiert werden sollen. Die symbolischen Namen (aus dem oben genannten Include-File) sind wohl eindeutig genug.

In der nächsten Zeile habe ich dem Window noch zwei Eigenschaften verpaßt. Smart Refresh heißt, daß der betroffene Inhalt des Fensters gerettet werden soll, wenn es von einem anderen ganz oder teilweise überdeckt wird. Sobald das Fenster wieder oben liegt oder der abgedeckte Teil wieder sichtbar wird, zeichnet Intuition den Fensterinhalt neu. Im Gegensatz dazu gibt es noch Simple Refresh. In diesem Fall meldet Intuition nur den »Schaden«. GEM kann übrigens nur letzteres.

Nach dem Öffnen des Fensters kann man im Fenster zeichnen. Mit Move( ) wird die Zeichenposition gesetzt, mit Text( ) ab dieser Stelle ein String ausgegeben. In beiden Fällen muß man den Rast-Port wissen. Graphics benötigt immer die Adresse des Rast-Ports, dem ein Window zugeordnet ist. Ein Rast-Port ist vereinfacht ausgedrückt eine Struktur, die die Zeichenbedingungen etwas ausführlicher beschreibt als ein Fenster. Seine Adresse kann man sich mit

```
move.l  wd_RPort(A1),A1
```

holen, wenn vorher der Window-Zeiger in A1 geladen wurde. Dahinter steckt immer eine Technik, die ich Ihnen an den folgenden Zeilen (Auszug aus dem Listing) verdeutlichen möchte:

```
move.l   windowptr,aØ          ;zeige auf Window-Struktur
move.l   wd_UserPort(aØ),aØ    ;nun auf Message-Port
move.b   MP_SIGBIT(aØ),d1      ;Anzahl Signal Bits -> d1
```

Unsere Window-Struktur im Listing ist nur eine Hilfskonstruktion. Nach dem Aufruf von OpenWindow erhalten wir in D0 einen Zeiger auf eine ähnliche Struktur, die Intuition für uns anlegt. Unsere Struktur können wir danach wegwerfen, ändern, den Speicherbereich anders belegen oder sie modifiziert für ein weiteres Fenster benutzen.

Wichtig ist, daß wir uns den Zeiger gut merken, hier in der Variablen »windowptr«. Die erste der drei Zeilen ist noch einfach: Der Zeiger wird in das Register A0 geladen. Die Konstante »wd_UserPort« ist das Offset vom Beginn der Window-Struktur auf ein Langwort innerhalb der Struktur. Dieses Langwort ist aber selbst auch nur ein Zeiger auf eine andere Struktur, nämlich den Window-User-Port. Innerhalb dieser Struktur gibt es ein Byte, in dem das Message-Port-Signal-Bit notiert ist. Genau das brauchen wir.

Auf diese Art greift man also auf einen der beiden IDCM-Ports zu. Intuition richtet automatisch zwei dieser Ports zu jedem Window ein. Der Empfangs-Port heißt User-Port, senden kann man über den WindowPort. In diesem Beispiel zwar überflüssig, aber um es mal zu zeigen: Benutzt man Wait( ), muß man sagen, bei welchen Signal-Bits (praktisch Semaphoren) man geweckt werden möchte.

Da zu jedem Port ein Signal-Bit gehört, müssen wir natürlich feststellen, welches unserer ist. Im Feld MP_SIGBIT steht, welches Bit das ist (als Bit-Nummer), die Funktion Wait( ) erwartet aber eine Maske in D0, in der genau dieses Bit gesetzt ist. Deshalb lade ich das Register D1 mit der Bit-Nummer, lade dann D0 mit 1 und schiebe nun mit LSL diese Eins auf den richtigen Platz. Sobald das Bit »klingelt«, wurde unser Window angesprochen. Da wir als Event nur CLOSEWINDOW zugelassen hatten, können wir uns weitere Tests sparen und mit dem Schließen des Windows sowie aller offenen Libs das Programm beenden.

Kapitel 9

Vom CLI-Task zum »Clickable Icon«

CLI-Task

Workbench-Task

Startup-Code

Icon-Editor

9.1 Programm-Betriebsarten

Fast alle Programme, die wir bisher geschrieben haben, konnten wir nur mit ihren Namen im CLI aufrufen. Das ist nicht die schlechteste Lösung, denn viele CLI-Kommandos sind auch nur Programme dieser Art, doch Sie wissen selbst, daß der Amiga mehr kann.

Sie wissen auch, daß jedes CLI (mit NEWCLI können Sie zusätzliche CLIs schaffen) ein Task ist. Unsere Programme waren für den CLI-Task, von dem aus sie aufgerufen wurden, praktisch nur Unterprogramme. Solange unser Programm lief, also zum Beispiel auf eine Eingabe wartete, war auch das CLI in diesem Unterprogramm und wartete.

Die nächsthöhere Stufe ist ein Programm, das im CLI mit »RUN Name« aufgerufen werden kann. Dieses Programm läuft dann wirklich als eigener Task, und das CLI ist wieder frei für andere Dinge.

Die höchste Stufe bilden dann Programme, die man von der Workbench aus starten kann, indem man einfach mit der Maus auf ihr Icon klickt. Ziel dieses Kapitels ist es, ein solches Programm zu erstellen, und natürlich zu zeigen, wie man dazu vorgehen muß.

Sozusagen bei der Gelegenheit sollen gleich zwei Fliegen mit einer Klappe erschlagen werden, sprich, es gibt da noch eine Art von Programmen, nämlich solche, die sowohl vom CLI als auch von der Workbench aus gestartet werden können. Ich glaube, wir sind uns einig, daß jedes Workbench-Programm auch unter dem CLI laufen sollte, folglich können wir auf die Lösung »Nur Workbench« getrost verzichten. Wie Sie gleich sehen werden, ist die damit erzielbare Einsparung auch nur minimal. Fassen wir zusammen. Es gibt:

1. CLI-Unterprogramme (Aufruf mit Namen)
2. CLI-Tasks (Aufruf mit RUN Namen)
3. Workbench-Tasks (Klick auf das Icon)
4. Die Kombination von 2. und 3.

Die Gruppen 1 und 2 unterscheiden sich nur minimal. Gruppe 1 benutzt für die Eingabe und Ausgabe das CLI-Window. Das Handle dafür wird mit der DOS-Funktion Input bzw. Output ermittelt. Gruppe 2 arbeitet nicht mit diesen Handles, sondern mit einem eigenen Fenster. Die Programme im Kapitel 5 gehören dazu. Probieren Sie es aus und starten diese Programme mit »RUN Name«. Achten Sie aber bitte darauf, daß Sie immer vorab erst mit der Maus das Fenster anklicken müssen, in dem Sie arbeiten wollen. Wenn Sie genau hinsehen, werden Sie feststellen, daß auf dem Schirm nach dem Aufruf auch »CLI2« steht. Praktisch heißt das: CLI 1 richtet für das Programm temporär (solange es läuft) ein neues CLI ein.

9.2 Der Startup-Code

Workbench-Programme (Gruppe 3) müssen eine Zusatzbedingung erfüllen. Sie dürfen nicht einfach loslaufen, wann Sie wollen, sondern müssen sozusagen auf die Starterlaubnis der Workbench warten. Deshalb müssen solche Programme zu Beginn auf eine Message (den Startbefehl) warten. Wenn Sie fertig sind, müssen Sie das der Workbench melden, indem Sie genau diese Message (die Sie sich gut gemerkt haben) an die Workbench zurückschicken.

Wird dasselbe Programm vom CLI aus aufgerufen, entfällt natürlich diese Geschichte. Das heißt auch, daß unser Programm unterscheiden muß, von wo es aufgerufen wurde.

Der Trick hinter der ganzen Geschichte ist der sogenannte Startup-Code. Der Name ist nicht ganz korrekt, hat sich jedoch so eingebürgert. Zum Start-Code gehört nämlich immer auch ein Ende-Code. Um nun beides in einen einzigen File packen zu können, den man bei HiSoft und SEKA als Include-File zuerst lädt und bei Metacomco mit dem Linker vor sein Programm setzt, kommt wieder ein kleiner Trick zum Tragen. Normalerweise hieße die Folge:
- Start-Code
- Unser Programmteil
- Ende-Code

Praktisch gehen wir aber so vor:

```
            Start-Code
            jsr _main
            End-Code

_main       Unser Programmteil
            rts
```

Sie wissen jetzt, warum ich in all meinen Listings die Label »_main« an den Anfang gesetzt habe (wenn Sie sie nicht sehen, _main steckt im Include-File OpenDOS.i). Bei Metacomco sollten Sie auf jeden Fall mit »_main« arbeiten, der Linker erwartet dies, wenn Sie »startup.o« einbinden.

Auch dürfte jetzt klar sein, daß unsere Programme immer mit einem schlichten aber wichtigen »rts« enden müssen. Doch nun zur Praxis. Bild 9.1 zeigt das Listing des Startups.

Im »ROM-Kernel-Manual, Libraries and Devices« finden Sie ein recht langes Assembler-Listing, das den Startup-Code für C-Programme bildet. Dieses Listing habe ich etwas umgestaltet und drastisch gekürzt. Wenn Sie noch einige der dort aufgeführten

Features einbauen wollen, zum Beispiel Alerts für den wohl sehr wahrscheinlichen Fall (???), daß sich die DOS-Lib nicht öffnen läßt, O.K., da ist die Quelle.

```
* startup.i

* Startup-Code fuer Assembler-Programme. Recht frei nach dem
* Beispiel im ROM-Kernel-Manual Libraries and Devices, aber so
* geschrieben, dass als Include-File brauchbar und auf das
* wirklich Notwendige reduziert
* -----------------------------------------------------------

        incdir   ":include/"

        include  "exec/exec_lib.i"
        include  "libraries/dosextens.i

        movem.l  dØ/aØ,-(sp)             ;rette Kommandozeile
        clr.l    _WBenchMsg              ;sicherheitshalber

* Teste, von wo wir gestartet wurden
* ---------------------------------
        sub.l    a1,a1                   ;a1=Ø = eigener Task
        CALLEXEC FindTask                ;wo sind wir?
        move.l   dØ,a4                   ;Adresse retten

        tst.l    pr_CLI(a4)              ;Laufen wir unter WB?    SEKA: $AC(a4)
        beq.s    fromWorkbench           ;wenn so

* Wir wurden vom CLI gestartet
* ---------------------------
        movem.l  (sp)+,dØ/aØ             ;Parms Kommandozeile holen
        bra      run                     ;und starten

* Wir wurden von  Workbench gestartet
* ----------------------------------
fromWorkbench
        lea      pr_MsgPort(a4),aØ       ;SEKA:    $5C(a4)
        CALLEXEC WaitPort                ;Warte auf Start-Message
        lea      pr_MsgPort(a4),aØ       ;sie ist da
        CALLEXEC GetMsg                  ;hole sie
        move.l   dØ,_WBenchMsg           ;immer Msg sichern!

        movem.l  (sp)+,dØ/aØ             ;bringe Stack i.O.

run     bsr.s    _main                   ;rufe unser Programm auf

        move.l   dØ,-(sp)                ;rette seinen Return-Code

        tst.l    _WBenchMsg              ;gibt's eine WB-Message
```

```
        beq.s     _exit                     ;nein: dann war's CLI

        CALLEXEC Forbid                     ;keine Unterbrechung jetzt
        move.l    _WBenchMsg(pc),a1         ;hole die Message
        CALLEXEC ReplyMsg                   ;und gib sie zurueck

_exit
        move.l    (sp)+,dØ                  ;hole Return-Code
        rts                                 ;das war's

_WBenchMsg        ds.l   1

        cnop      Ø,2
```

Bild 9.1: Der Startup-Code

Kern der Angelegenheit sind diese Zeilen:

```
        sub.l     a1,a1                     ;a1=Ø = eigener Task
        CALLEXEC FindTask                   ;wo sind wir?
        move.l    dØ,a4                     ;Adresse retten
        tst.l     pr_CLI(a4)                ;Laufen wir unter WB?
        beq.s     fromWorkbench             ;wenn so
```

Die Exec-Funktion »FindTask« findet die Adresse einer Task-Kontroll-Struktur. Normalerweise übergibt man der Funktion die Adresse eines Strings mit dem Task-Namen im Register a1. Ist dieser Zeiger Null, erhält man die Adresse des eigenen Tasks. Nun muß ich leider gestehen, daß hier »Task« nicht korrekt ist. Genauer behandeln wir dieses Thema aber erst im Kapitel 14, jetzt nur soviel: Wir laufen unter einem DOS-Prozeß. Ein Prozeß ist so etwas Ähnliches wie ein Task, nur höherwertiger. »FindTask« gibt deshalb die Adresse unseres PLB (Prozeß-Leit-Block) zurück. Wenn Sie sich diese Struktur ansehen möchten: sie steht im Include-File »include/libraries/dosextens.i«.

Innerhalb dieser Struktur gibt es den Eintrag mit dem Offset »pr_CLI«. Das soll heißen »Pointer (Zeiger) auf den Command Line Interpreter«. Dieser Zeiger ist Null, wenn wir unter der Workbench laufen.

Also geht es in diesem Fall zum Label »fromWorkbench« und da steht:

```
fromWorkbench
        lea       pr_MsgPort(a4),aØ
        CALLEXEC WaitPort                   ;Warte auf Start-Message
        lea       pr_MsgPort(a4),aØ         ;sie ist da
        CALLEXEC GetMsg                     ;hole sie
        move.l    dØ,_WBenchMsg             ;immer Msg sichern!
```

Im PLB beim Offset »pr_MsgPort« steht die Adresse, die Exec-Funktion »WaitPort« sehen will. Dieser Aufruf läßt unseren Task warten, bis er an die Reihe kommt. Stellen Sie sich das so vor: Es gibt eine Liste aller laufenden Tasks. Exec sorgt dafür, daß einer nach dem anderen für eine gewisse Zeit an die Reihe kommt, denn praktisch kann immer nur ein Task laufen (wir haben nur einen 68000 im Amiga). Seinen Startbefehl erhält der Task über diesen Message-Port, genauer: nur die Nachricht, daß eine Message da ist. Deshalb muß man mit »GetMsg« diese Nachricht aus dem Port lesen. Sie steht danach im Register d0. Da wir diese Nachricht noch brauchen, sichern wir sie in der Variablen »_WBenchMsg«.

Wenn unser Task unter der Workbench läuft, und er »geweckt« wurde, sind jetzt praktisch diese Zeilen interessant:

```
run     bsr.s    _main             ;rufe unser Programm auf
        CALLEXEC Forbid            ;keine Unterbrechung jetzt
        move.l   _WBenchMsg,a1     ;hole die Message
        CALLEXEC ReplyMsg          ;und gib sie zurueck
```

Mit »bsr _main« wird nun endlich unser Programmteil aufgerufen. Danach erfolgen die Rückzugsgefechte, das, was ich in der Einleitung den Ende-Code genannt habe. Wir müssen uns bei der Workbench ordnungsgemäß abmelden, was an sich dadurch geschieht, daß wir mit »ReplyMsg« die ursprünglich beim Start erhaltene Message zurückgeben. Da auch andere Tasks theoretisch zur selben Zeit auf den PLB zugreifen können, könnte es sein, daß unsere Nachricht nicht ankommt oder schlimmer, das totale Chaos ausbricht.

In einem Multitaskingsystem, in dem verschiedene Tasks auf gemeinsame globale Variable zugreifen können, gibt es deshalb immer einen Mechanismus, der einem Task für eine gewisse Zeit das alleinige Zugriffsrecht sichert. Diese Funktion heißt beim Amiga Forbid (verbiete mir Störungen). Genaugenommen ist diese Funktion recht gefährlich, denn sie schaltet das Multitasking aus. Es bleibt ausgeschaltet, solange der Task läuft oder bis er Wait (warte auf Nachricht) oder Permit aufruft. Da wir nach dem Forbid nur noch die Reply-Message zurückgeben und dann enden, ist das »Forbidding« hier vertretbar (und notwendig sowieso). Nach dem »bsr _main« hatten wir noch mit

```
        move.l   dØ,-(sp)
```

den Return-Code unseres Programms gesichert. Es ist unsere Sache, was wir da zurückgeben wollen. Üblich ist Null für keinen Fehler. Dieses »d0« müssen wir natürlich vor dem RTS wieder vom Stack holen. Genau so hatten wir gleich zu Anfang des Programms mit

```
        movem.l  dØ/aØ,-(sp)       ;rette Kommandozeile
        clr.l    _WBenchMsg        ;sicherheitshalber
```

die Länge und Adresse einer eventuellen Kommandozeile gesichert und die Variable _WBenchMsg auf Null gesetzt. Das Aufräumen des Stacks mit

```
        movem.l  (sp)+,dØ/aØ
```

erfolgt dann jeweils entweder im CLI- oder im Workbench-Zweig. Schauen wir uns zum Schluß noch an, was denn tatsächlich im Fall von CLI geschieht, so bleibt:

```
run     bsr.s    _main               ;rufe unser Programm auf
        rts                          ;das war's
```

Wenn Sie diesen Quelltext nun assemblieren, müßte das mit einer Ausnahme ohne Fehler über die Bühne gehen. Die Ausnahme ist das fehlende Label »_main«.

Nun speichern Sie diesen File bitte unter dem Namen »startup.i« (notfalls finden Sie ihn auch auf der Diskette zu diesem Buch), im nächsten Abschnitt wollen wir ihn praktisch erproben.

9.3 Multitasking-Demo

Um nun einmal zu zeigen, daß einer unserer Tasks tatsächlich ständig läuft und etwas tut, wollen wir nun ein Programm schreiben, das in einem Window ständig den noch freien RAM anzeigt. Das Listing dazu bringt Bild 9.2.

```
        opt      l-                  ;nicht linken!

* free_ram

        incdir   ":include/"

        include  startup.i           ;oder wie Sie ihn nennen

        include  intuition/intuition.i
        include  intuition/intuition_lib.i
        include  exec/memory.i
        include  graphics/graphics_lib.i
        include  libraries/dos_lib.i

GRAFIC  macro
        move.l   windowptr,a1        ;Adresse Window-Struktur
        move.l   wd_RPort(a1),a1     ;von da auf Rast Port
        CALLGRAF \1                  ;Grafik-Funktion
```

```
        endm

_main

* DOS-Library oeffnen
* -------------------
        lea      dosname,a1
        moveq    #Ø,dØ
        CALLEXEC OpenLibrary
        tst.l    dØ
        beq      abbruch
        move.l   dØ,_DOSBase            ;Basis-Zeiger sichern

* Intuition Library oeffnen:
* -------------------------
        lea      intname,a1
        moveq    #Ø,dØ
        CALLEXEC OpenLibrary
        tst.l    dØ
        beq      closedos
        move.l   dØ,_IntuitionBase    ;Basis-Zeiger sichern

* Graphics Library oeffnen
* -----------------------
        lea      grafname,a1
        moveq    #Ø,dØ
        CALLEXEC OpenLibrary
        tst.l    dØ
        beq      closeint
        move.l   dØ,_GfxBase          ;Basis-Zeiger sichern

* Window oeffnen
* -------------
        lea      windowdef,aØ         ;zeige auf Window-Struktur
        CALLINT  OpenWindow           ;oeffne Window
        tst.l    dØ                   ;ging was schief?
        beq      closegraf               ;wenn ja
        move.l   dØ,windowptr            ;Window-Zeiger sichern

* Hauptschleife
* ------------
loop    moveq    #MEMF_PUBLIC,d1         ;freien RAM
        CALLEXEC AvailMem                ;einlesen
        move.l   dØ,d2                   ;nach d2
        lea      buffer,aØ               ;in Hex-String
        bsr      hex

        moveq    #8Ø,dØ                  ;X-Position fuer Text
        moveq    #19,d1                  ;Y
        GRAFIC   Move                    ;Move TO X,Y

        lea      buffer,aØ               ;Text-Adresse
```

```
        addq.l   #2,a0                  ;die 2 ersten Nibbles sind
        moveq    #6,d0                  ;eh 0, restliche 6 reichen
        GRAFIC   Text                   ;Text zeichnen

        move.l   windowptr,a0       ;Von unserem Window
        move.l   wd_UserPort(a0),a0 ;den Empfangsport
        CALLEXEC GetMsg             ;testen
        tst.l    d0                 ;Haben wir Post?
        bne      fini               ;kann nur CLOSEWINDOW sein

        move.l   #25,d1             ;25/50 = 1/2 Sekunde
        CALLDOS  Delay              ;warten
        bra      loop               ;und von vorn

fini    move.l   d0,a1              ;Message ist in d0
        CALLEXEC ReplyMsg           ;antworten

closewindow
        move.l   windowptr,a0       ;Fenster zu
        CALLINT  CloseWindow

closegraf
        move.l   _GfxBase,a1        ;Die Libs schliessen:
        CALLEXEC CloseLibrary

closeint
        move.l   _IntuitionBase,a1
        CALLEXEC CloseLibrary

closedos
        move.l   _DOSBase,a1
        CALLEXEC CloseLibrary

abbruch
        moveq    #0,d0              ;keinen Fehler melden
        rts                         ;und Ende

* Konvertiere d2.l in ASCII-String ab (a0)
* ---------------------------------------
hex     moveq   #8-1,d1         ;nun alle Nibble
next    rol.l   #4,d2           ;hole 1 Nibble
        move.l  d2,d3           ;nach d3 retten
        and.b   #$0f,d3         ;maskiere es
        add.b   #48,d3          ;in ASCII wandeln
        cmp.b   #58,d3          ;ist es >9 ?
        bcs     out             ;wenn nicht
        addq.b  #7,d3           ;sonst muss es A-F sein
out     move.b  d3,(a0)+        ;1 Zeichen abspeichern
        dbra    d1,next         ;next nibble
        rts

W_Gadgets equu   WINDOWDRAG!WINDOWDEPTH!WINDOWCLOSE
```

```
W_Extras  equ    SMART_REFRESH!ACTIVATE

W_Title dc.b     ' Freier RAM z.Z. ',Ø

windowdef
        dc.w     2ØØ,2Ø                ;links, oben
        dc.w     22Ø,24                ;Breite, Hoehe
        dc.b     -1,-1                 ;Pens des Screen
        dc.l     CLOSEWINDOW           ;einziges IDCMP Flag
        dc.l     W_Gadgets!W_Extras    ;Window Flags
        dc.l     Ø                     ;keine User-Gadgets
        dc.l     Ø                     ;keine User-Checkmark
        dc.l     W_Title               ;Titel des Window
        dc.l     Ø                     ;kein eigener Screen
        dc.l     Ø                     ;keine Super Bitmap
        dc.w     1ØØ,2Ø                ;Min. Groesse
        dc.w     64Ø,2ØØ               ;Max.
        dc.w     WBENCHSCREEN          ;Use Workbench Screen

intname          INTNAME               ;Namen der Libs aus Makros
grafname         GRAFNAME
dosname          DOSNAME

buffer  ds.b     8

_IntuitionBase  ds.l    1              ;Speicher fuer die Zeiger
_GfxBase        ds.l    1
_DOSBase        ds.l    1
windowptr       ds.l    1
```

Bild 9.2: Ein Programm, das immer den freien Speicher anzeigt

Zuerst möchte ich Ihnen etwas Taktik verdeutlichen, nämlich hiermit:

```
* DOS-Library oeffnen        :   beq      abbruch
* Intuition Library oeffnen:     beq      closedos
* Graphics Library oeffnen :     beq      closeint
* Window oeffnen             :   beq      closegraf

closewindow: Window schließen
closegraf  : Grafic-Lib schließen
closeint   : Intuition-Lib schließen
closedos   : DOS-Lib schließen
abbruch    : Programm-Ende
```

Das Problem ist, daß ich immer nur die Libraries schließen darf, die ich vorher auch geöffnet hatte. Wenn man nun mehrere Libs der Reihe nach öffnet, dann muß man natürlich wissen, welche im Fehlerfall schon offen waren, nun also zu schließen sind.

Die Lösung ist ganz simpel. Die Schließ-Routinen werden in der umgekehrten Reihenfolge, wie die Open-Routinen geschrieben. Im Fehlerfall springt man dann nur noch zur Close-Routine, die nach der Close-Routine steht, deren Öffnen fehlschlug. Spielen Sie im obigen Beispiel mal einige Fälle durch; ich hoffe, es stimmt.

Nachdem nun alle Libraries, die wir hier benötigen, geöffnet sind, kann es losgehen. Der Kern des Programms steckt in diesen Zeilen:

```
loop    moveq    #MEMF_PUBLIC,d1         ;freien RAM
        CALLEXEC AvailMem                ;einlesen
        move.l   dØ,d2                   ;nach d2
        lea      buffer,aØ               ;in Hex-String
        bsr      hex
```

Wir holen uns den freien Speicher, indem wir der Exec-Funktion »AvailMem« die Konstante »MEMF_PUBLIC« übergeben. Diese Konstante ist im Include-File »include/exec/memory.i« mittels EQU definiert. Sie werden dort auch noch einige andere Parameter finden. Machen Sie sich einmal den Spaß und ändern das Programm so, daß auch die Größen von Fast-RAM, Chip-RAM usw. angezeigt werden.

Nun folgt die Hex-Konvertierung, die Sie schon aus Kapitel 5 kennen und die Ausgabe des Textes, wie im Beispiel von Kapitel 8. Neu ist nur der Makro GRAFIC, der mir einiges an Tipperei erspart. Anders ist auch das Warten auf ein Intuition-Event gelöst. Auch hier erwarten wir gemäß unserer Window-Definition nur eines, nämlich CLOSEWINDOW.

```
        move.l   windowptr,aØ            ;Von unserem Window
        move.l   wd_UserPort(aØ),aØ      ;den Empfangsport
        CALLEXEC GetMsg                  ;testen
        tst.l    dØ                      ;Haben wir Post?
        bne      fini                    ;kann nur CLOSEWINDOW sein
```

Wie Sie sehen, reicht es, einfach den Message-Port zu lesen. Ist der »Briefkasten« leer, ist d0 null. Normalerweise ist diese Methode des sogenannten Pollings unschön, doch hier geht es, weil wir im Fall von keine Nachricht mit diesen Zeilen weitermachen:

```
        move.l  #25,d1                   ;25/5Ø = 1/2 Sekunde
        CALLDOS Delay                    ;warten
        bra     loop                     ;und von vorn
```

Mit dem Aufruf der Delay-Routine geben wir nämlich den anderen Tasks für eine halbe Sekunde Zeit (aus CPU-Sicht die reinste Ewigkeit), selbst etwas zu tun. Wenn alle halbe Sekunde der neue Stand des RAM angezeigt wird, dürfte es wohl reichen. Wer schneller informiert sein will, kann die Zeit natürlich kürzer wählen.

Wurde das Close-Gadget des Fensters angeklickt, dann haben wir eine Message erhalten und es erfolgt ein Sprung zum Label »fini«. Dort steht

```
fini    move.l   dØ,a1                  ;Message ist in dØ
        CALLEXEC ReplyMsg               ;antworten
```

und das möchte ich Ihnen noch einmal besonders ans Herz legen. Wir müssen Intuition auf jede Nachricht antworten! Das geschieht ganz einfach durch den Vermerk »zurück an Absender«, sprich, wir senden die eben erhaltene Nachricht zurück.

Ist das Programm (fehlerfrei) assembliert und heißt das Ergebnis »free_ram«, dann können Sie jetzt ins CLI gehen und »run free_ram« tippen. Nun sollte das Fenster mit der Anzeige erscheinen. Um nun ein CLI-Kommando ausführen zu können, müssen Sie zuerst irgendwo im CLI-Fenster klicken. Geben Sie nun DIR ein, und Sie werden sehen, wie sich die Speicheranzeige ständig ändert, solange DIR läuft. Einen kleinen Fehler hat unser Programm noch. Wir wollen in sein Fenster nichts eingeben, warum also ist das Fenster aktiv, und warum müssen wir erst das CLI-Fenster anklicken? Die Lösung ist recht einfach. Ändern Sie die Zeile

```
W_Extras  equ   SMART_REFRESH!ACTIVATE
```

in

```
W_Extras  equ   SMART_REFRESH
```

Damit ist das Fenster nicht mehr aktiv, sein Titel wird dann grau geschrieben. Sie können es natürlich anklicken, wenn Ihnen die aktive Form besser gefällt.

9.4 Icons und der Icon-Editor

Um ein Programm, das mit einem ordentlichen Startup-Code versehen ist, von der Workbench aus starten zu können, benötigt es nur noch ein Icon. Damit ein Icon sichtbar wird, muß es a) vorhanden sein (logisch) und b) in einem Directory stehen, das selbst ein Icon hat, sprich als Schublade sichtbar ist. Am einfachsten stellen Sie eine solche Schublade her, indem Sie auf der Workbench die Schublade »Empty« duplizieren. Sie können aber auch im CLI einfach tippen:

```
        copy empty.info  test.info
```

Kehren Sie nun zur Workbench zurück, so sehen Sie das neue Icon nicht. Schließen Sie dann das Disk-Fenster und öffnen es wieder. Nun ziehen Sie die Empty-Schublade etwas weg und die Test-Schublade wird sichtbar.

Jetzt brauchen wir ein Programm-Icon. Dazu nehmen Sie am besten auch ein vorhandenes Icon, allerdings ist nicht jedes geeignet. Der Amiga kennt verschiedene Typen von Icons. Welche das sind und was sie für eine Bedeutung haben, erfahren Sie automatisch, wenn Sie den Icon-Editor starten. Für uns ist wichtig zu wissen, daß Programme vom Typ TOOL sein müssen.

Geeignet ist zum Beispiel IconED selbst. Nehmen wir an, Sie haben das Directory (die Schublade) »test« schon erstellt und unser Programm hieße »free_ram«. Dann kopieren Sie zuerst das Programm mit

```
copy free_ram :test/free_ram
```

Nun kopieren Sie ein Icon dazu (IconEd steckt im System-Ordner)

```
copy :system/iconed.info :test/free_ram.info
```

Nun sollten Sie auf der Workbench in der Schublade »test« ein Icon finden, das aussieht wie das von IconEd, aber den Titel »free_ram« zeigt. Das können Sie nun getrost anklicken, »free_ram« wird starten.

Wenn Sie jetzt Ihrem Icon ein eigenes Aussehen verpassen wollen, rufen Sie IconEd auf. Im Disk-Menü wählen Sie LOAD und tippen dann in den Text-Requester

```
:test/free_ram
```

sprich immer den vollen Pfadnamen. Das Editieren ist simpel und im Prinzip selbsterklärend. Probieren Sie einfach die verschiedenen Menü-Punkte aus. Wichtig zu wissen ist: Um ein Icon zeichnen/ändern zu können, müssen Sie immer das Menü Color anwählen und daraus die passende Farbe. Radieren können Sie mit der Hintergrundfarbe. Gezeichnet/radiert wird mit der Maus. Die linke Taste drückt den Stift auf das »Papier«.

Sie können auch auf den Kopiervorgang ganz verzichten und ein Icon selbst im Editor erstellen. Sie müssen dann nur im Save-Requester den korrekten Namen eingeben, in unserem Beispiel also wieder

```
:test/free_ram
```

Ansonsten keine Sorge. Es können zwar die unmöglichsten Icons entstehen, aber editiert wird immer nur der Info-File. Ihrem Programm passiert nichts.

9.5 Langworte in Dezimalstrings wandeln

Bei Betrachtung der Speicheranzeige hatte meine Frau Probleme, die Hex-Anzeige sagte ihr nichts, dezimal wollte sie es haben. Da kann man bekanntlich nichts machen, also hier die Lösung mit Bild 9.3. Für die Bindec-Routine bräuchten wir eine Langwort-Division. Die finden Sie in nahezu jedem 68000-Buch, deshalb hier zur Abwechslung einmal etwas anderes.

Das Verfahren ist uralt, ich habe es schon im BASIC-Interpretern der ersten Stunde gesehen. Auf einem 68000 läuft es besonders schnell, da dieser es mit seinen raffinierten Adressierungsarten direkt unterstützt. Das Prinzip heißt Division durch fortlaufende Subtraktion. Nehmen wir an, die Zahl hieße 321. Dann kann ich davon 3 mal 100 subtrahieren, beim vierten Versuch gibt es einen Unterlauf. Ich zähle nun, wie oft ich bis zum Unterlauf brauche (4 mal), subtrahiere davon wieder 1 und habe das erste Digit (die 3). Nun muß ich allerdings auch auf die verbleibende Zahl wieder 100 addieren, blieben 21. Von dieser 21 subtrahiere ich nun 10er-weise. Das bringt beim dritten Versuch den Unterlauf, minus 1 ergibt das Digit 2.

Unser Langwort ist nun allerdings für Zahlen von gut +/- 2 Milliarden gut, womit unsere Zahlenreihe nicht bei 100 sondern bei 10 000 000 beginnen muß. Diese Kolonne finden Sie in der Tabelle. Merke: Tabellenzugriff ist immer schneller als die rechnerische Ermittlung der Zahl. Neu ist noch, daß ich hier auf Vorzeichen achte. Im Fall negativer Zahlen wird eine positive erzwungen (neg.l) und in d3 dieser Fall notiert. Später, nachdem die führenden Nullen durch Blanks ersetzt wurden, wird dann ein Minus-Zeichen in den Puffer geschrieben. Beachten Sie bitte, daß Sie hier zehn Stellen mittels _LVOWrite ausgeben sollten.

```
* Konvertiere d2.l -> Dec-String ab (aØ)
decl    clr.b   d3                  ;Ø = positive Zahl
        tst.l   d2                  ;Zahl positiv
        bpl     plus                ;wenn so
        neg.l   d2                  ;sonst wandeln
        move.b  #1,d3               ;markiert negative Zahl
plus    moveq   #7,dØ               ;8 Digits konnvertieren
        lea     buffer+1,aØ         ;+1 f. Platz f. Vorzeichen
        lea     pwrof1Ø,a1          ;Tabelle
next    moveq   #'Ø',d1             ;Fange mit Digit 'Ø' an
dec     addq    #1,d1               ;Digit + 1
        sub.l   (a1),d2             ;noch drin?
        bcc.s   dec                 ;wenn so
        subq    #1,d1               ;korrigiere Digit
        add.l   (a1),d2             ;den auch
        move.b  d1,(aØ)+            ;Digit -> Buffer
        lea     4(a1),a1            ;next power_1Ø
```

```
        dbra    dØ,next             ;for 8 Digits
        lea     buffer,aØ           ;Nun Ø-Unterdr. u. Vorz.
rep     move.b  #' ',(aØ)+          ;fuehrende Nullen
        cmp.b   #'Ø',(aØ)           ;  durch Blanks
        beq     rep                 ;    ersetzen
        tst.b   d3                  ;war Zahl negativ?
        beq     done                ;wenn nicht
        move.b  #'-',-1(aØ)         ;sonst - vorsetzen
done    rts
pwrof1Ø dc.l    1ØØØØØØØ
        dc.l    1ØØØØØØ
        dc.l    1ØØØØØ
        dc.l    1ØØØØ
        dc.l    1ØØØ
        dc.l    1ØØ
        dc.l    1Ø
        dc.l    1
```

Bild 9.3: Routine zur dezimalen Darstellung von »Longs«

Kapitel 10

Der Befehlssatz des 68000 im Überblick

Internas

Hintergrundwissen

In diesem Kapitel soll der Befehlssatz des 68000 im Überblick vorgestellt werden. Zu jedem einzelnen Befehl finden Sie die syntaktischen Formen und die erlaubten Adressierungsarten im Anhang A1. Dort ist auch beschrieben, welche Operandenlängen (Byte, Word, Long) jeweils zulässig sind. Hier geht es primär um die Thematik »was gibt es, und wofür braucht man es«.

10.1 Transfer-Befehle

Befehl	Bedeutung
EXG	Austausch von Registerinhalten
LEA	Laden eines Registers
LINK	Lokalen Stack aufbauen
MOVE	Übertragen (kopieren) von Daten
MOVEA	Übertragen (kopieren) von Adressen
MOVEM	Übertragen (kopieren) mehrerer Register
MOVEP	Übertragen (kopieren) von Daten zur Peripherie
MOVEQ	Übertragen (kopieren) von Konstanten »Quick«
PEA	Adresse auf den Stack bringen
SWAP	Vertauschen der Worte eines Registers
UNLK	Abbau des lokalen Stacks (siehe LINK)

Auffallend sind sicherlich die vielen Varianten des MOVE-Befehls. Hier sollten Sie einmal Ihren Assembler testen. Gute Assembler akzeptieren auch ein MOVE, wo man eigentlich MOVEA hätte schreiben müssen. Sehr gute Assembler geben sogenannte Warnings aus, wenn Sie nicht optimal programmieren, hier also anstatt eines zulässigen MOVEQ nur ein einfaches MOVE schreiben würden.

10.1.1 LINK und UNLINK

Besonders erwähnenswert sind sicherlich die Befehle LINK und UNLK (Unlink). Mit diesen Befehlen ist der 68000 besonders gut auf die Aufgabenstellung von Hochsprachen-Compilern vorbereitet. Hier ergibt sich immer das Problem, daß in Prozeduren und Funktionen lokale Variable geschaffen werden müssen, die nur solange existieren sollen, wie die Prozedur (Funktion) aktiv ist. Typisch werden deshalb solche Variablen auf dem Stack abgelegt.

Ideal ist es nun, wenn man mit nur einem Befehl den passenden Stackbereich reservieren und dann später auf genauso einfache Art wieder freigeben kann. Genau das bieten die Befehle LINK und UNLK. Wenn nun eine Prozedur (Unterprogramm) eine weitere Prozedur aufruft und diese dann eine dritte usw., und jede dieser Prozeduren mit dem eigenen lokalen Stack arbeitet, dann entsteht sozusagen eine verkettete Liste, englisch »linked list«. Daher rühren auch die Namen LINK und UNLK. Schauen wir uns jetzt einmal an, wie das funktioniert. Die Syntax des Befehls lautet

```
LINK An,#Adreßdistanz
```

Ein Beispiel:

```
LINK A6,#30
```

In diesem Fall wird zuerst A6 auf dem Stack abgelegt, praktisch der Befehl

```
move.l  a6,-(sp)    (Schritt 1)
```

ausgeführt. Nun wird der Stackpointer in das soeben gerettete Register kopiert, sprich

```
move.l  sp,a6       (Schritt 2)
```

Zuletzt wird die Adreßdistanz auf den Stackpointer addiert, das heißt

```
add.l   #Adr_Dist,sp
```

Damit hätten wir den lokalen Stack für ein Unterprogramm. Üblicherweise wählt man die Adreßdistanz negativ, da der Stack bekanntlich zu fallenden Adressen hinwächst. Mit UNLK wird der ursprüngliche Zustand wiederhergestellt. Praktisch wirkt UNLK wie

```
move.l  a6,a7
move.l  (sp)+,a6
```

Für das Hauptprogramm oder allgemein das aufrufende Unterprogramm hat somit das Adreßregister und der Stackpointer wieder seinen ursprünglichen Wert.

Zu beachten ist noch, daß das Adreßregister nach dem LINK-Befehl eine Kopie des Stackpointers hält. Somit kann das Unterprogramm sehr einfach auf Daten des aufrufenden Programms zugreifen, wenn dieses die Daten vorher auf den Stack gepackt hat.

10.2 Arithmetische Befehle

Befehl	Bedeutung
ADD	Addition von Daten
ADDA	Addition von Adressen
ADDI	Addition einer Konstanten
ADDQ	Addition einer Konstanten »Quick«
ADDX	Addition mit Übertrag-Bit
CLR	Löschen eines Operanden
CMP	Vergleich zweier Daten
CMPA	Vergleich zweier Adressen
CMPI	Vergleich mit einer Konstanten
CMPM	Vergleich zweier Daten im Speicher
DIVS	Division mit Vorzeichen
DIVU	Division ohne Vorzeichen
EXT	Vorzeichenrichtige Erweiterung
MULS	Multiplikation mit Vorzeichen
MULU	Multiplikation ohne Vorzeichen
NEG	Negation
NEGX	Negation mit X-Bit
SUB	Subtraktion von Daten
SUBA	Subtraktion von Adressen
SUBI	Subtraktion einer Konstanten
SUBQ	Subtraktion einer Konstanten »Quick«
SUBX	Subtraktion mit X-Bit (Borgen)
TST	Teste Operanden gegen Null
ABCD	Addition von BCD-Zahlen
NBCD	Negation von BCD-Zahlen
SBCD	Subtraktion von BCD-Zahlen

Auch hier können gute Assembler wieder glänzen. Zumindest sollten sie mit ADD einverstanden sein, wenn eigentlich ADDA oder ADDI erforderlich ist. Sinngemäßes gilt für CMP und SUB.

Besonders zu loben ist hier der 68000 wegen zweier Eigenschaften. Zuerst: Die arithmetischen Operationen sind auf einer Breite von 32 Bit möglich. In diesem Sinn ist also

der 68000 ein echter »32-Bitter«. Sein Datenbus ist zwar nur 16 Bit breit, was dazu führt, daß Langworte in »zwei Portionen« transportiert werden, das interessiert aber nur sekundär. Primär ist wichtig, daß damit mathematische Operationen wesentlich einfacher zu programmieren sind als auf CPUs, die nur 16 oder gar nur 8 Bit breite Operanden zulassen. Sekundär sollte man natürlich darauf achten, daß die Daten so lange als möglich in Registern gehalten werden, denn dann entfällt auch der relativ zeitaufwendige Transfer über den Datenbus. Bei der hohen Geschwindigkeit des 68000 sollte man das allerdings nicht überbewerten. Nur in sehr rechenintensiven Routinen könnte damit etwas erreicht werden.

10.2.1 BCD-Arithmetik

Die BCD-Arithmetik des 68000 wird jeder schätzen, der schon einmal auf einer anderen CPU so etwas programmieren mußte. Eine BCD-Ziffer steht immer in einem Halb-Byte (4 Bit). Da sich damit bekanntlich die Zahlen 0 bis 15 darstellen lassen, hier aber nur 0 bis 9 gültig sind, gibt es beim Überlauf einige Probleme. Bei anderen CPUs muß man diesen Fall mit Hilfe des sogenannten Half-Carry-Flags testen; hier kann man einfach addieren. Da als Operandengröße immer nur Byte zugelassen ist, gibt es den Überlauf nach 99. Dieser geht aber automatisch in das X-Flag und wird auch automatisch immer mitaddiert. Hier ein Beispiel für die Addition zweier 6-stelliger Zahlen in je 3 Byte.

Zahl	Wert	im Speicher auf Adressen
1	123456	12 auf 1001, 34 auf 1002, 56 auf 1003
2	654321	65 auf 2001, 43 auf 2002, 21 auf 1003

Wie üblich muß man rechts (bei den Einern) mit der Addition beginnen. Daher ist als Speicheradressierungsart hier auch nur »ARI mit Predekrement« erlaubt. Das Programm sähe dann so aus:

```
move   #1004,a1
move   #2004,a2
move   #4,CCR

ABCD   -(a1),-(a2)
ABCD   -(a1),-(a2)
ABCD   -(a1),-(a2)
```

Zu beachten ist dabei dreierlei:

1. Wegen des Predekrements muß das Byte mit dem »Einer« auf einer ungeraden Adresse liegen.

2. Um nicht beim ersten Mal ein zufälliges X-Flag mitzuaddieren, muß man es löschen.
3. Um ein Null-Ergebnis erkennen zu können, sollte man vorher das Z-Flag setzen.

Die Punkte 2 und 3 lassen sich mit der Anweisung »move #4,CCR« sehr einfach zusammen erledigen. Eines bleibt Ihnen allerdings nicht erspart. Sie müssen schon garantieren, daß die Zahlen BCD-Zahlen sind (Abfrage beim Laden). Größere Werte als 9 werden nämlich schlicht falsch addiert.

10.3 Logische Befehle

Befehl	Bedeutung
AND	Logisch UND
ANDI	Logisch UND mit einer Konstanten
EOR	Logisch XOR
EORI	Logisch XOR mit einer Konstanten
NOT	Logisch NICHT (Einerkomplement)
OR	Logisch ODER
ORI	Logisch ODER mit einer Konstanten

Zu diesen Befehlen könnte man vielleicht nur noch anmerken (Sie wissen es schon), daß sie bitweise wirken.

10.4 Bit-Befehle

Befehl	Bedeutung
BCHG	Ändere (kippe) ein Bit
BCLR	Lösche ein Bit
BSET	Setze ein Bit
BTST	Prüfe ein Bit
TAS	Teste und setze Bit 7 eines Byte-Operanden

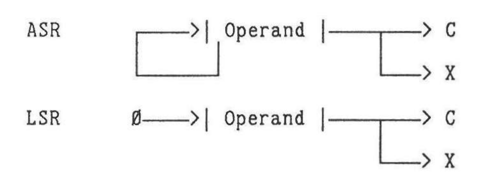
Die Bit-Befehle halten immer den vorherigen Zustand im Z-Flag fest, führen also zwei Operationen aus. Diese Eigenschaft, und ganz besonders die Fähigkeit von TAS sind ein gutes Beispiel für eine besondere Fähigkeit des 68000, nämlich die Unterstützung

von Multitasking. Mit dem TAS-Befehl wird Bit 7 eines Byte-Operanden im Speicher abgefragt und das Ergebnis wie üblich im Z-Flag notiert. Dann wird eine 1 in Bit 7 geschrieben. Die ganze Folge, also Lesen des Operanden, Abfrage und Zurückschreiben ist unteilbar, sprich kann nicht durch einen Interrupt unterbrochen werden. Solche unteilbaren Befehle sind in einem Multitasking-System sehr wichtig. Hier wären zum Beispiel Aufgaben wie Prozeßumschaltung, Synchronisation von Prozessen und ähnliche zu nennen.

10.5 Schiebe- und Rotierbefehle

Befehl	Bedeutung
ASL	Arithmetische Verschiebung nach links
ASR	Arithmetische Verschiebung nach rechts
LSL	Logische Verschiebung nach links
LSR	Logische Verschiebung nach rechts
ROL	Rotieren links herum
ROR	Rotieren rechts herum
ROXL	Rotieren mit X-Bit links
ROXR	Rotieren mit X-Bit rechts

Anzumerken ist hierzu, daß mit einem Befehl um bis zu 31 Bit geschoben/rotiert werden kann. Andere CPUs schaffen mit einer Anweisung immer nur ein Bit. Zwischen ASL und LSL besteht praktisch kein Unterschied, wohl aber einer zwischen ASR und LSR, wie das folgende Bild zeigt:



Die Zeichnung soll zeigen, daß beim ASR das Vorzeichen-Bit immer wiederhergestellt, praktisch also nicht geschoben wird. Beim LSR wird hingegen ein Null-Bit nachgeschoben. Beim Linksschieben wird sowohl beim ASL als auch beim LSL ein Null-Bit rechts eingespeist.

10.6 Programmsteuer-Befehle

Befehl	Bedeutung
Bcc	Verzweige bedingt
BRA	Verzweige immer
BSR	Verzweige zu einem Unterprogramm
CHK	Checke Datenregister gegen Grenze 0 und andere
DBcc	Bedingte Schleife
JMP	Sprung zu einer Adresse
JSR	Sprung zu einem Unterprogramm
NOP	Keine Operation
RESET	Rücksetzen der Peripherie
RTE	Rückkehr von einer Exception
RTR	Rückkehr mit Laden der Flags
RTS	Rückkehr aus einem Unterprogramm
Scc	Setze ein Byte bedingt
STOP	Halte Programm an
TRAP	Gehe in Exception
TRAPV	Gehe in Exception, wenn V-Flag gesetzt

Zu beachten wäre hier der Unterschied zwischen den Verzweigungs- und den Sprungbefehlen. Erstere sind immer relativ zum aktuellen PC, allerdings auf einen Adreßbereich von + -32 Kbyte begrenzt. Die Sprungbefehle JMP und JSR reichen über den vollen Adreßbereich von 16 Mbyte.

CHK testet ein Datenregister gegen zwei Grenzen, nämlich 0 und eine im Operanden angegebene Grenze. Spricht der Test an, wird eine Exception ausgelöst. Damit läßt sich sehr einfach eine Bereichsprüfung, zum Beispiel von Array-Indizes, realisieren.

Wenn Sie nun richtig mitgezählt haben, waren das 56 Befehle. Im Vergleich mit anderen CPUs ist das relativ wenig, doch das täuscht. Die meisten der 12 Grundadressierungsarten können auf den Quell- und den Zieloperanden angewendet werden, womit sich über 1000 Varianten ergeben. Andere CPUs geben einigen dieser Varianten eigene Befehlsnamen, womit zwar nicht die Leistung aber das Lernpensum des Programmierers gesteigert wird.

10.7 Hintergrundwissen

In diesem Abschnitt soll etwas Hintergrundwissen zum 68000 vermittelt werden. Das brauchen Sie zwar nicht unbedingt zum Programmieren, aber sicherlich ist Ihnen auch wohler, wenn Sie wissen, warum Sie was tun.

Wahrscheinlich haben Sie beim Lesen des Buchs auch schon gemerkt, daß ich ein 68000-Verehrer bin. Das ist nun nicht so aus der Luft gegriffen, sondern basiert auf langjähriger Erfahrung in der Programmierung anderer CPUs, nach dem Motto: Wer das Schlechte kennt, weiß das Gute zu schätzen.

Der 68000 ist der erste Mikroprozessor, dessen Befehlssatz an den der Mini-Computer angelehnt ist. Wenn man bedenkt, daß so moderne Rechner wie der Amiga mit seinem Hauptspeicher von 1-4 Mbyte (und 256 Kbyte im ROM) den Minis der 70er Jahre inzwischen überlegen ist, ein 68020-System inzwischen schon eine VAX in Teilbereichen schlägt, dann ist dieser Befehlssatz auch die einzige Alternative. Leistungsfähige Rechner brauchen nämlich sehr komplexe System-Software, die nur sicher und effektiv zu erstellen ist, wenn die CPU die entsprechenden Grundlagen bietet.

10.7.1 Die innere Struktur des 68000

Prinzipiell besteht eine CPU immer aus einem Steuerwerk und einem Rechenwerk, man sagt auch »Control Logic« und »Arithmetic Logic Unit« (ALU). Das Steuerwerk besteht aus

- Befehls-Register und
- Befehls-Decoder

Der Befehls-Decoder gibt seine Ergebnisse an die Ausführungseinheit, wo er zum Beispiel ALU-Funktionen oder Registerauswahl anstößt. Kern der Sache ist nun die Befehlsdecodierung. Die ersten Mikroprozessoren, wie zum Beispiel der legendäre F8, waren im Prinzip nichts weiter als programmierbare Logikbausteine. Bestimmte Bitmuster an den Eingängen erzeugen da andere Bitmuster an den Ausgängen. In diesem Sinn waren Befehle auch nur Bitmuster, die per Hardware (mit vielen Gattern) decodiert wurden. Die Technik läßt sich bei einer 8-Bit-CPU vielleicht heute noch vertreten, führt aber bei einem so mächtigen Befehlssatz wie dem des 68000 schnell in eine Sackgasse. Die Hardware wird dann nämlich bald unüberschaubar, verbraucht sehr viel Platz und ist kaum noch zu ändern. Der Ausweg ist der sogenannte *Mikro-Code*.

Vereinfacht ausgedrückt heißt das, daß die Befehle des 68000, wie wir sie kennen, aus Sicht der CPU schon eine Hochsprache sind. Die CPU übersetzt mittels eines Programms diese von außen kommenden Makro-Befehle in eine Folge von internen Mikro-Befehlen. Das Steuerwerk der CPU ist demnach nicht als Hardware-Logik sondern als Programm ausgelegt. Dieses Programm befindet sich in einem ROM-Bereich auf

dem CPU-Chip. Da aber letztendlich doch Hardware in der CPU anzusprechen ist, ergibt sich ein Problem. Es sind bei jedem Befehl sehr viele »Bits« zu setzen, zum Beispiel schon 64, wenn zwei Register angesprochen werden. Macht man nun den Mikro-Code sehr schmal (zum Beispiel 4 Bit breit), benötigt man sehr viele Mikro-Zyklen, also viel Zeit zur Auflösung eines Makro-Befehls. Man spricht hier vom vertikalen Mikro-Code. Macht man den Mikro-Code horizontal breiter, entstehen weniger Zyklen. Jetzt ist aber der Decoder (zeit)aufwendiger. Man kann nun zwischen beiden Arten (vertikal oder horizontal) einen Kompromiß finden oder - noch besser - beide miteinander kombinieren.

Genau das macht der 68000. Es gibt einen Mikro-Code-ROM (Befehlsbreite 9 Bit) und einen Nano-Code-ROM (70 Bit). Die Mikro-Codes sind im Prinzip nur Zeiger auf die Nano-Codes, womit sich eine sehr schnelle Decodierung ergibt, ähnlich, als wenn man in einem Buch nicht alle Seiten durchblättert, sondern zuerst im Inhaltsverzeichnis nachsieht.

Trotzdem ist die reine Hardware-Logik schneller, weil es dort prinzipiell keine Suchzeiten gibt. Um diesen Nachteil auszugleichen, hat man sich etwas einfallen lassen, was *Prefetch* heißt. Prefetch bedeutet soviel wie »vorab holen«. Ein Befehl kann aus bis zu fünf Worten bestehen. Der allgemeine Zyklus lautet

- Holen
- Decodieren
- Ausführen

Überlappt man diese Vorgänge, spart man natürlich Zeit. Praktisch holt der 68000 bei der Abarbeitung eines Befehls schon das nächste Befehlswort und das darauffolgende. Dann wird immer ein neues Wort eingelesen, wenn/während eins abgearbeitet wird.

10.7.2 User- und Supervisor-Modus

Wie schon geschildert, gibt es beim 68000 den User- und den Supervisor-Modus. Der Vorteil dieser Trennung ist klar. Kernroutinen des Betriebssystems können nicht durch einen Fehler in einem User-Programm gestört werden. Nur dieses ermöglicht uns »Usern« überhaupt, solchen Fehlern auf die Spur zu kommen. Denn wie soll zum Beispiel ein Debugger uns die Registerinhalte anzeigen, wenn die dafür erforderlichen Betriebssystem-Routinen durch das fehlerhafte Programm zerstört wurden?

In welchem Modus sich der 68000 befindet, entscheidet nur ein Bit, nämlich das S-Bit im Statusregister. Wenn der Amiga durch einen Kaltstart (Einschalten) oder Warmstart (Reset-Taste) anläuft, befindet er sich automatisch im Supervisor-Modus. Der Übergang vom Supervisor- in den User-Modus kann erfolgen durch

- RTE
- Änderung des Supervisor-Bits (MOVE #K,SR / ANDI #K,SR u.a.)

Vom User-Modus in den Supervisor-Modus gelangt man durch
- Interrupt
- Trap-Befehl
- Exception (zum Beispiel Adreß-Error)

Im User-Modus stehen alle acht Datenregister, die sieben Adreßregister A0 bis A6, der Stackpointer (USP) und der PC zur Verfügung. Keinerlei Zugriff besteht auf den Supervisor-Stackpointer (SSP).

Eingeschränkt ist der Zugriff auf das Statusregister. Dieses besteht aus dem Supervisor-Byte, auch System-Byte genannt, und dem User-Byte (CCR = Condition Code Register). Auf das CCR besteht voller Zugriff, auf das System-Byte kann im User-Modus nur lesend zugegriffen werden.

Trace-Bit

Einen besonderen Komfort bietet das Trace-Bit. Ist dieses Bit (Bit 15 im Statusregister) gesetzt, geht der 68000 nach jedem Befehl in eine Exception, sprich springt zur Adresse, die im Trace-Vektor (Vektor 9 auf Adresse $24) eingetragen ist. Damit ist eine Einzelschrittbearbeitung möglich. Die Routine, auf die der Trace-Vektor zeigt, kann dann zum Beispiel die Registerinhalte anzeigen. Anders ausgedrückt: Der schwierigste Teil eines Debuggers ist beim 68000 schon eingebaut. Sie können sich sicherlich vorstellen, daß Tracing ohne dieses Feature recht aufwendig zu programmieren ist. Man muß dann nämlich im zu testenden Code immer auf den nächsten Befehl einen Sprung zur Trace-Routine legen, dann diesen Befehl wieder durch das Original ersetzen, den Trace-Sprung verlegen usw. Dazu muß man natürlich wissen, wieviel Bytes jeder Befehl belegt, sprich einen Disassembler mitlaufen lassen.

10.7.3 Die Exceptions

Wir haben nun Exceptions schon so oft angewandt, ohne es zu merken (Exec arbeitet damit), daß wir uns dieses wichtige Feature einmal genauer ansehen sollten.

Eine Exception ist eine Ausnahme, wenn man einmal nur das Wort als solches übersetzt. In den Ausnahmezustand kann der 68000 auf drei Arten gebracht werden:
1. Durch externe Signale (Interrupt, Busfehler, Reset)
2. Durch Fehler (zum Beispiel Adreßfehler)
3. »Mit Absicht«

Der dritte Fall wird häufig von Exec genutzt, der mit den sogenannten Trap-Befehlen Exceptions auslöst. Wenn wir selbst damit arbeiten wollen, sollten wir besser einen Exception-Vektor von Exec anfordern. Andernfalls gibt es bestimmt Zusammenstöße mit dem Betriebssystem. Wer vom Atari ST oder Macintosh her kommend den ein-

fachen und häufigen Umgang mit den Traps vermißt, sei darauf hingewiesen, daß in einem Multitasking-System das Betriebssystem doch etwas komplizierter ist. Reset oder Interrupt sind auch noch einfach zu erklären, dazu muß man nur die entsprechenden Eingänge des 68000 ansteuern.

Bus-Error

Der Busfehler ist schon etwas komplizierter. Bei den »68000ern« gibt es Hardwarebeschaltungen (MMU), die zuerst dafür sorgen, daß die Adressen, die der 68000 generiert, auch die richtigen Speicherchips ansprechen. So eine MMU hat immer einen Fehler-(Fault)Ausgang, der dann aktiviert wird, wenn eine nicht existierende Adresse angesprochen wird.

Was passiert bei einer Exception?

Die Frage kann man à la Radio Erivan beantworten: Es kommt darauf an... Tatsächlich gibt es zwei Klassen von Exceptions. Allgemein wird immer der Programmzähler (PC) und das Statusregister (SR) auf dem Stack abgelegt. Bei einem Bus-oder Adreß-Error kommen noch drei Informationen hinzu, nämlich
- der Code des gerade abgearbeiteten Befehls
- die Adresse, auf die gerade zugegriffen wurde
- das Super-Statuswort

Im Super-Statuswort sind die Bits 0 bis 4 relevant und zwar

Bit 0 ... 2:	Funktions-Code
Bit 3:	0 = Gruppe 2, 1 = Gruppe 1
Bit 4:	0 = Schreibzyklus wurde unterbrochen
	1 = Lesezyklus

Nun wäre wieder einiges zu erklären. Die Funktions-Codes sind drei Ausgänge des 68000, mit der die CPU der MMU anzeigt, was sie gerade tut. Im wesentlichen erfolgt hier die Unterscheidung in Zugriffe auf Anwender-Daten, Anwender-Programm, Supervisor-Daten und Supervisor-Programm. Dahinter steckt, daß die CPU natürlich weiß, ob sie im Supervisor- oder User-Modus ist, oder ob der PC auf ein Befehlswort oder ein Datum zeigt.

Die Gruppen 0, 1, und 2 haben folgenden Sinn. Überlegen Sie einmal, was passiert, wenn der 68000 in einer Exception-Bearbeitung ist und gerade dann noch eine Exception auftritt!

Nun, für so etwas gibt es Prioritäten. Die höchste Priorität hat die Gruppe 0, dann folgen 1 und 2. Gruppiert wird so:

Gruppe 0	Reset Bus-Error Adreß-Error
Gruppe 1	Trace Interrupt Illegaler Befehl Trap $Axxx, Trap $Fxxx Privilegverletzung
Gruppe 2	Trap #n Trapv CHK Division durch 0

Man kann die Liste auch in einem Stück sehen. Danach hat dann Reset die höchste und »Division durch Null« die niedrigste Priorität. Tritt nun zum Beispiel während einer Exception eine solche mit höherer Priorität auf, wird das laufende Programm unterbrochen, die Routine mit der höheren Priorität abgearbeitet und dann die unterbrochene Routine fortgesetzt.

Exceptions beim Amiga

Wie schon geschildert, wird nach einer Exception einiges an Informationen auf dem Stack gesichert und dann der PC mit der Adresse geladen, die im zugehörigen Vektor steht. Das wirkt dann wie ein Sprung zu dieser Adresse. Deshalb sollte in jedem Vektor schon etwas eingetragen sein, damit der 68000 nicht »in den Wald läuft«. Im Vektor 0 steht, welchen Wert der Stackpointer nach Reset einnehmen soll, im Vektor 1 der dann erste Stand des PCs. Diese Vektoren sind obligatorisch und müssen im Einschalt- oder Reset-Augenblick vorhanden sein. Da beim Einschalten ein RAM natürlich leer ist, muß per Hardware dafür gesorgt werden, daß dann »etwas ROM« auf diesen Adressen liegt. Die übrigen Vektoren kann nun jeder Systemprogrammierer nach Gusto belegen. Alle wird er kaum benötigen. Die dann freien Vektoren sollten nun aber zumindest alle mit einer (ein und derselben) Adresse geladen werden. Die Routine auf dieser Adresse kann schlicht mit RTE beginnen (und enden) oder eine kleine Meldung der Art »Vektor Nummer X nicht belegt« ausgeben. Anstatt dieser Meldung zeigt nun der Amiga seine wenig geliebte Guru-Meditation.

Kapitel 11

Datenstrukturen des Amiga

Tricks mit Makros

und

Datenstrukturen

Der Schlüssel zur Amiga-Programmierung

11.1 Datenstrukturen, der Schlüssel zur Amiga-Programmierung

Die erste Datenstruktur hatten wir im Kapitel 8, Bild 8.1 schon kennengelernt. Hier noch einmal mit Bild 11.1 die Window-Struktur:

```
windowdef
        dc.w    200,50              ;links,oben
        dc.w    300,100             ;Breite, Hoehe
        dc.b    -1,-1               ;Pens des Screen
        dc.l    CLOSEWINDOW         ;einziges IDCMP Flag
        dc.l    W_Gadgets!W_Extras  ;Window Flags
        dc.l    0                   ;keine User-Gadgets
        c.l     0                   ;keine User-Checkmark
        dc.l    W_Title             ;Titel des Window
        dc.l    0                   ;kein eigener Screen
        dc.l    0                   ;keine Super Bitmap
        dc.w    100,20              ;Min. Groesse
        dc.w    640,200             ;Max.
        dc.w    WBENCHSCREEN        ;Use Workbench Screen
```

Bild 11.1: Die NewWindow-Struktur

Wenn man einmal von den Pflichtübungen zu Beginn eines Programms (Libraries öffnen) und denen zum Schluß (Libraries schließen) absieht, dann besteht ein typisches Amiga-Programm eigentlich nur aus zwei sich immer wiederholenden Dingen, nämlich
- Datenstruktur definieren
- Funktion mit der Struktur als Parameter aufrufen

Im Beispiel von Kapitel 8 wollten wir ein Intuition-Window öffnen. Das Öffnen selbst wurde mit einer Zeile erledigt, die Arbeit steckte in den Zeilen von Bild 11.1. Mit dem Schreiben von »Hello World!« haben wir es uns noch relativ einfach gemacht. Wir hätten auch einen anderen Font (Schrifttyp) nehmen können, doch dazu hätten wir noch schreiben müssen:

```
MyFont  dc.l    font_name
        dc.w    TOPAZ_SIXTY
        dc.b    FS_NORMAL
        dc.b    FPF_ROMFONT
```

Das wäre noch eine Struktur. Wir hätten auch einen eigenen Screen definieren können, auch das erfordert eine Struktur (Thema kommt noch), wir möchten mit Pull-down-Menüs arbeiten (viele Strukturen) oder oder oder...

Wie auch immer, was auch immer Sie programmieren wollen, an den Strukturen kommen Sie nicht vorbei. Leider sind diese Strukturen zum Teil recht komplex, auf jeden Fall sind sie sehr zahlreich und ein kleiner Fehler (dc anstatt dc.l) führt mindestens zu seltsamen Ergebnissen, meistens aber zum Absturz.

11.2 Include-Files

Die wichtigsten Strukturen finden Sie »assembler-gerecht« im Anhang, alle aufzuführen würde das Buch sprengen. Sie finden aber alle Strukturen in den Include-Files, leider aber dort in einer sehr schwer verständlichen Weise dargestellt und - noch schlimmer - in einer Form, die für Assembler-Programme ziemlich unpraktisch ist (für die meisten C-Programme übrigens auch).

Das Problem: Die Strukturen wurden ursprünglich in C geschrieben und dann in Assembler umgesetzt. Faul, wie gute Programmierer von Natur aus sind (bin ein sehr guter Programmierer), haben sie dafür natürlich Makros benutzt, und das sieht dann so aus, wie in Bild 9.2.

```
STRUCTURE   macro               ;Krücke für manche Assembler
\1          set     Ø           ;damit das Kind einen Namen
Offset      set     \2          ;bekommen kann und bei Ø
                                ;anfängt
            endm


BYTE        macro
\1          equ     Offset
Offset      set     Offset+1    ;der LC zählt in Bytes
            endm

WORD        macro
\1          equ     Offset
Offset      set     Offset+2    ;Ein Wort hat 2 Byte
            endm


LONG        macro
\1          equ     Offset
```

```
Offset      set     Offset+4        ;Ein Langwort hat 4 Byte
            endm

ULONG       macro                   ;Unsigned Long auch
\1          equ     Offset
Offset      set     Offset+4
            endm

APTR        macro
\1          equ     Offset
Offset      set     Offset+4        ;A Pointer ist auch lang
            endm

LABEL       macro
\1          equ     Offset
            endm
```

Bild 11.2: Einige Makros sinngemäß wie in »exec/types.i«

11.3 Aufbau von Strukturen (?) mit Makros

Wenn Sie sich die Namen in Bild 11.2 so anschauen und vielleicht schon einmal in der Amiga-Dokumentation geblättert haben, dann kommt Ihnen da einiges bestimmt bekannt vor. Es sind Datentypen, wie sie im Amiga-C verwendet werden. In Assembler müssen wir uns um gewisse Spitzfindigkeiten des C-Compilers aber nicht kümmern. Ein Byte ist für uns ein Byte. Ob das später mal eine vorzeichenbehaftete Zahl aufnehmen soll oder nicht, ist uns egal. Wir brauchen deshalb nicht zwischen UBYTE (Unsigned Byte) und BYTE zu unterscheiden. Sinngemäßes gilt für ULONG und LONG.

Im HiSoft-Assembler ist die Sache viel einfacher, weil der die Direktive RS kennt. Da schreibt man dann anstatt

```
        WORD  xxx
```

einfach

```
        xxx   rs.w    1
```

Doch schauen wir uns doch einmal so eine Makro-Entwicklung an. Da steht zum Beispiel:

```
STRUCTURE    macro                    ;Krücke für manche Assembler
\1           set     Ø                ;damit das Kind einen Namen
Offset       set     \2               ;bekommen kann und bei Ø
                                      ;anfängt
             endm
```

Diesen Makro rufe ich auf mit

```
        STRUCTURE NewWindow,Ø
```

Dann wird daraus

```
NewWindow      set     Ø
Offset         set     Ø
```

Zwei Konstanten sind entstanden, beide haben den Wert 0. Nun rufe ich wieder einen Makro auf, jetzt aber diesen:

```
WORD         macro
\1           equ     Offset
Offset       set     Offset+2        ;Ein Wort hat 2 Byte
             endm
```

Der Aufruf erfolgt mit

```
    WORD nw_LeftEdge
```

Dann entsteht daraus

```
nw_LeftEdge    equ     Offset
Offset         set     Offset+2
```

Folge der Übung: »nw_LeftEdge« hat jetzt den Wert 0 (den alten Wert von Offset), und Offset selbst steht nun auf 2. Das kann ich nun fortsetzen. Die komplette Übung zeigt Bild 11.3.

Auf diese Art hat man eine Tabelle von Konstanten angelegt. Wenn Sie diese Tabelle einmal mit Bild 11.1 vergleichen, so fällt Ihnen sicherlich auf, daß hier symbolische Namen für die einzelnen Elemente eingesetzt wurden. Ferner dürfte klar sein, daß die Offsets den Typen entsprechen, also zum Beispiel für ein Langwort hier 4 Byte definiert werden.

Solche Offset-Tabellen treffen Sie in den Include-Files sehr häufig an und besonders in »intuition.« auch sehr zahlreich. Sind das nun die Strukturen, über die wir hier reden? Die klare Antwort: Nein!

```
Direktive                          Wert von
                               Label      Offset
----------------------------------------------

 STRUCTURE NewWindow,Ø          Ø          Ø

    WORD nw_LeftEdge            Ø          2
    WORD nw_TopEdge             2          4
    WORD nw_Width               4          6
    WORD nw_Height              6          8
    BYTE nw_DetailPen           8          9
    BYTE nw_BlockPen            9          1Ø
    ULONG nw_IDCMPFlags        1Ø          14
    LONG nw_Flags              14          18
    APTR nw_FirstGadget        18          22
    APTR nw_CheckMark          22          26
    APTR nw_Title              26          3Ø
    APTR nw_Screen             3Ø          34
    APTR nw_BitMap             34          38
    WORD nw_MinWidth           38          4Ø
    WORD nw_MinHeight          4Ø          42
    WORD nw_MaxWidth           42          44
    WORD nw_MaxHeight          44          46
    WORD nw_Type               46          48

    LABEL nw_SIZE              48          48
```

Bild 11.3: Eine Offset-Tabelle

Eine Offset-Tabelle ist keine Datenstruktur!

Auch wenn in den Metacomco-Include-Files die Gebilde mit dem Wort STRUCTURE beginnen, so sind das keine Strukturen. Dieser Makro-Name STRUCTURE ist (gelinde gesagt) unklug gewählt. In den Datenstrukturen kann ich etwas hineinschreiben, in ein Offset (praktisch in eine Konstante) wohl kaum. Was nutzt mir also das Gebilde? Schauen wir uns dazu Bild 11.4 an.

```
* window2

* In diesen Files stecken diverse Deklarationen und Makros.
* Schauen Sie mal rein!

        incdir  ":include/"
```

```
        include intuition/intuition.i
        include intuition/intuition_lib.i
        include exec/exec_lib.i
        include graphics/graphics_lib.i

* Intuition Library oeffnen:
* --------------------------
        lea     intname,a1
        moveq   #Ø,dØ
        CALLEXEC OpenLibrary
        tst.l   dØ
        beq     abbruch
        move.l  dØ,_IntuitionBase       ;Basis-Zeiger sichern

* Graphics Library oeffnen
* -------------------------
        lea     grafname,a1
        moveq   #Ø,dØ
        CALLEXEC OpenLibrary
        tst.l   dØ
        beq     closeint
        move.l  dØ,_GfxBase            ;Basis-Zeiger sichern

* Window oeffnen
* --------------
        jsr     InitWindow              ;initialisiere NewWindow

        lea     NewWindow,aØ            ;zeige auf Window-Struktur
        CALLINT OpenWindow              ;oeffne Window
        tst.l   dØ                      ;ging was schief?
        beq     closegraf               ;wenn ja
        move.l  dØ,windowptr            ;Window-Zeiger sichern

* Text im Fenster zeichnen
* ------------------------
       moveq   #1ØØ,dØ                  ;X-Position
       moveq   #5Ø,d1                   ;Y
       move.l  windowptr,a1             ;Via Window-Zeiger
       move.l  wd_RPort(a1),a1          ;  Rast-Port-Adresse holen
       CALLGRAF Move                    ;Funktion Move to X,Y

       move.l  windowptr,a1             ;brauche wieder Rast-Port
       move.l  wd_RPort(a1),a1
       lea     msg,aØ                   ;Adresse Text
       moveq   #msglen,dØ               ;seine Laenge
       CALLGRAF Text                    ;und ausgeben

* Auf Event warten (kann hier nur WINDOWCLOSE sein)
* -------------------------------------------------
        move.l  windowptr,aØ            ;zeige auf Window-Struktur
        move.l  wd_UserPort(aØ),aØ      ;nun auf Message-Port
        move.b  MP_SIGBIT(aØ),d1        ;Anzahl Signal Bits -> d1
```

```
        moveq.l #1,d0                   ;in Maske
        lsl.l   d1,d0                   ;       konvertieren
        CALLEXEC Wait                   ;Schlaf gut!

* Fenster schliessen
* ------------------
        move.l  windowptr,a0            ;wir sind wieder wach
        CALLINT CloseWindow             ;Fenster zu

* Libraries schliessen
* --------------------
closegraf
        move.l  _GfxBase,a1
        CALLEXEC CloseLibrary

closeint
        move.l  _IntuitionBase,a1
        CALLEXEC CloseLibrary

abbruch
        move.l  #0,d0                   ;oder normales Ende
        rts

W_Gadgets equu  WINDOWSIZING!WINDOWDRAG!WINDOWDEPTH!WINDOWCLOSE
W_Extras  equ   SMART_REFRESH!ACTIVATE

W_Title dc.b    'Fenster-Titel',0

***************************************************************
NewWindow       ds.b    nw_SIZE     ;Puffer f. Windowstruktur *
***************************************************************

InitWindow
        lea     NewWindow,a0                      ;Fuelle Struktur
        move.w  #200,nw_LeftEdge(a0)
        move.w  #50,nw_TopEdge(a0)
        move.w  #300,nw_Width(a0)
        move.w  #100,nw_Height(a0)
        move.b  #0,nw_DetailPen(a0)
        move.b  #1,nw_BlockPen(a0)
        move.l  #W_Title,nw_Title(a0)
        move.l  #W_Gadgets!W_Extras,nw_Flags(a0)
        move.l  #CLOSEWINDOW,nw_IDCMPFlags(a0)
        clr.l   nw_FirstGadget(a0)
        clr.l   nw_CheckMark(a0)
        clr.l   nw_Screen(a0)
        clr.l   nw_BitMap(a0)
        move.w  #100,nw_MinWidth(a0)
        move.w  #20,nw_MinHeight(a0)
        move.w  #640,nw_MaxWidth(a0)
        move.w  #200,nw_MaxHeight(a0)
        move.w  #WBENCHSCREEN,nw_Type(a0)
```

```
        rts

intname         INTNAME                 ;Name Intuition Lib (via Makro)
grafname        GRAFNAME                ;Name Graphics Lib


msg             dc.b    'Hello, World! '
msglen          equ     *-msg

_IntuitionBase  ds.l    1               ;Speicher fuer Zeiger
_GfxBase        ds.l    1
windowptr       ds.l    1
```

Bild 11.4: Die Nutzung der Offset-Tabellen

11.4 Anwendung von Offset-Tabellen

Das Listing entspricht im wesentlichen dem von Bild 8.1, die Wirkung ist sogar absolut gleich. Im Absatz »Window öffnen« ist diese Zeile hinzugekommen:

```
        jsr     InitWindow              ;initialisiere NewWindow
```

Hier steckt nun das Neue. Wir brauchen zum Öffnen des Windows eine Struktur mit Werten, wie denen von Bild 11.1. Diese Struktur muß im RAM stehen und dafür brauchen wir einen Speicherbereich. Dieser Puffer wird mit

```
NewWindow       ds.b    nw_SIZE         ;Puffer f. Windowstruktur
```

geschaffen. Damit hätten wir einen leeren Puffer mit einer Größe von nw_SIZE (48 Byte). Sie wissen jetzt, warum in Bild 11.2 die Größe bestimmt wurde. Nun müssen wir diesen Puffer mit Daten füllen. Da uns nur Offsets zur Verfügung stehen, stellen wir mit

```
        lea     NewWindow,a0
```

das Register a0 auf den Beginn des Puffers und nutzen die schon oft genutzte Adressierungsart »ARI mit Offset« für Befehle wie

```
        move.w  #200,nw_LeftEdge(a0)
        move.w  #50,nw_TopEdge(a0)
        move.w  #300,nw_Width(a0)
        move.w  #100,nw_Height(a0)
```

Damit hätten wir die linke, obere Ecke sowie Breite und Höhe des Fensters bestimmt. Gefällt Ihnen das? Mir nicht! Den gleichen Zweck hätten wie nämlich auch mit einem schlichten

```
        dc.w  200,50,300,100
```

erreicht. Weitere dc-Anweisungen hätten den Puffer genauso initialisiert, wie es die weiteren Move-Befehle tun. Zum Schluß stehen in beiden Fällen die richtigen 48 Byte im Puffer. Den brauchen wir wohl, doch die Move-Befehle sind jetzt zusätzlich vorhanden. Diese Befehle verbrauchen natürlich auch Speicher, hier 18 x 6 = 108 Byte, und (viel schlimmer) sie kosten Laufzeit.

Das kommt also dabei heraus, wenn man C-Strukturen so einfach in Assembler übersetzt. Nun können natürlich kluge Leute argumentieren, daß dieses Verfahren andere Vorteile hat. Sie wissen ja schon, daß die NewWindow-Struktur nach dem Öffnen des Fensters nicht mehr gebraucht wird, weil Intuition daraus eine eigene und größere Struktur aufbaut. Wir können also die NewWindow-Struktur mit anderen Daten füllen und sie für ein zweites Fenster nochmals verwenden. Aber dafür braucht man Variable, und die hätte man nur bei dieser Konstruktion. Stimmt das? In C ja, in Assembler nicht! Was hindert uns, so etwas wie das in Bild 11.5 zu schreiben?

```
W_Title_1 dc.b    'Fenster-Titel 1',0
W_Title_2 dc.b    'Fenster-Titel 2',0

windowdef
pos       dc.w    200,50              ;links, oben
          dc.w    300,100             ;Breite, Hoehe
          dc.b    -1,-1               ;Pens des Screen
          dc.l    CLOSEWINDOW         ;einziges IDCMP Flag
          dc.l    W_Gadgets!W_Extras  ;Window Flags
          dc.l    0                   ;keine User-Gadgets
          dc.l    0                   ;keine User-Checkmark
titel     dc.l    W_Title_1           ;Titel des Window
          dc.l    0                   ;kein eigener Screen
          dc.l    0                   ;keine Super Bitmap
          dc.w    100,20              ;Min. Groesse
          dc.w    640,200             ;Max.
          dc.w    WBENCHSCREEN        ;Use Workbench Screen
```

Bild 11.5: Mit Labels werden dc-Strukturen flexibel

Nun, nichts hindert uns. »dc« heißt zwar »definiere Konstante«, aber das dürfen Sie nicht zu wörtlich nehmen. Sie können die Werte nachträglich noch ändern. Wir können also wie üblich ein Fenster öffnen. Danach brauchen wir ein zweites, das soll jetzt aber woanders liegen und einen anderen Titel haben. Also schreiben wir:

```
move    #300,pos
move    #100,pos+2
move.l  #W_Title_2,titel
```

Wollen Sie noch mehr ändern, so sehen Sie einfach noch ein paar mehr Labels vor. Sind Sie dazu zu faul (wie ich in diesem Beispiel), hilft auch Abzählen. Wenn der erste Wert die Label »pos« hat und »pos« zwei Byte belegt, dann beginnt der nächste Wert eben bei »pos+2«. Nun können wir wieder OpenWindow aufrufen, und das geht so:

```
lea     windowdef,a0            ;zeige auf Window-Struktur
CALLINT OpenWindow              ;oeffne Window
tst.l   d0                      ;ging was schief?
beq     close_1                 ;wenn ja
move.l  d0,windowptr_2          ;Window-Zeiger sichern
```

Achten Sie auf den berühmten kleinen Unterschied! Wir haben ein neues Window, und den Zeiger darauf müssen wir natürlich in einer eigenen Variablen abspeichern, hier »windowptr_2«.

Noch ein Unterschied: Ging beim Öffnen von Window 2 etwas schief, dürfen Sie nicht zu »closeint« springen, sondern zur Zeile, ab der Window 1 geschlossen wird. Darauf müßte dann »closeint« folgen.

Wenn Sie nun mit mehreren Windows experimentieren (diese Übung empfehle ich Ihnen), können Sie natürlich ein Fenster nach dem anderen öffnen und dann im nächsten Block »Waits« mit den zugehörigen Window-Pointern aufrufen. In diesem Fall können Sie die Fenster auch nur in dieser Reihenfolge schließen. Praktischer macht sich da die GetMsg/ReplyMsg-Lösung aus dem Kapitel 9. Letzter Tip: Sie sollten in extra Variablen notieren, ob ein Fenster auf oder zu ist. Sie dürfen nämlich ein Fenster, das zu ist, nicht mehr ansprechen.

Zum Schluß aber doch ein positiver Hinweis zu den Offset-Tabellen. Wie schon gesagt, baut Intuition aus unserer NewWindow-Stuktur eine größere Struktur im RAM auf. Um auf Elemente dieser Struktur zugreifen zu können, müssen wir natürlich die Offsets kennen. Wir haben sie schon benutzt, hier ein Beispiel:

```
move.l  windowptr,a0            ;zeige auf Window-Struktur
move.l  wd_UserPort(a0),a0      ;nun auf Message-Port
```

Sie erinnern sich? In dieser Struktur (und anderen dieser Art) steckt noch mehr. Deshalb wollen wir uns nach diesem Schnellkurs im nächsten Kapitel mit solchen Dingen etwas genauer befassen.

11.5 BPTR und BSTR

In den Makros der Include-Files oder in anderen Quellen werden Sie des öfteren auf die Begriffe BPTR und BSTR stoßen. Dabei handelt es sich um einen BCPL-Pointer bzw. um einen BCPL-String. Aha! Nun, BCPL ist eine C-ähnliche Sprache, in der die Amiga-Software zum Teil noch entwickelt wurde. Da BPCL für Rechner konzipiert wurde, die mit Langwort-Adressierung arbeiten, gibt es ein Problem, denn bekanntlich ist der 68000 eine Byte-Maschine, sprich, er kann jedes Byte adressieren, BPCL aber nur jedes vierte.

Deshalb muß ein BPTR immer auf eine Langwortgrenze zeigen. Wenn diese Pointer in Datenstrukturen auftauchen, muß diese Struktur auf »lang« justiert werden, was Sie mit »cnop 0,4« (SEKA: align 4) erreichen können. Gleiches gilt für BSTR. BSTR zeigt auf das erste Byte eines Strings. Dieses Byte hält die Stringlänge, die folgenden Bytes sind der Text an sich.

Falls Sie die Pointer selbst anwenden wollen, müssen Sie sie umrechnen, sprich einfach den BTR mit 4 multiplizieren. In Assembler geschieht dies am schnellsten mittels Linksschieben um 2.

Ein APTR ist übrigens ein ganz gewöhnlicher Pointer (eine Adresse), die Sie als Long so nehmen, wie sie ist.

Kapitel 12

Intuition komplett

Screens

Windows

Fonts

Events

Menüs

Gadgets

12.1 Screens

Der Amiga vermag beliebig viele Screens zu öffnen. Na, sagen wir fast beliebig viele, ein Screen kostet nämlich auch Speicher.

Ein Screen ist ein sogenannter virtueller Bildschirm, also ein scheinbarer, der sich aber prinzipiell wie ein echter Schirm verhält. Er kann also farbig oder monochrom sein, verschiedene Größen haben, verschiedene Auflösungen und andere unterschiedliche Eigenschaften. Mehrere dieser Screens können nun gleichzeitig auf dem Hardware-Bildschirm dargestellt werden.

Im Gegensatz zu Windows können diese Screens sich aber nicht überlappen, dürfen nur untereinander (nicht nebeneinander) liegen und können mit der Maus auch nur vertikal verschoben werden.

Ein Screen kann nun wieder mehrere Windows enthalten. Er prägt auch die Eigenschaften des Windows.

Screen öffnen

Generell öffnet sich ein Screen so wie ein Window. Wir brauchen eine Struktur (hier NewScreen), einen Zeiger auf diese Struktur und eine Funktion, die natürlich _LVOOpenScreen heißt. Als Ergebnis bekommen wir einen Zeiger auf eine Struktur zurück, die im Anhang abgedruckt und erklärt ist. Wichtig ist, daß wir uns diese Adresse merken und - ganz wichtig - diese Adresse auch in das Window (alle Windows) eintragen. Die Kernroutine in puncto Screen lautet also:

```
lea     NewScreen(pc),a0        ;Zeige auf Struktur
CALLINT OpenScreen              ;Open Screen
tst.l   d0                      ;ging was schief?
beq     closegfx                ;wenn ja
move.l  d0,Screen               ;in Window eintragen!
```

Den weiteren Anfang des Programms in Bild 12.1 kennen Sie schon. Beachten Sie jedoch bitte zwei Änderungen in der Window-Struktur: Zum einen ist das Label Screen hinzugekommen, auf diese Stelle wird die Screen-Struktur-Adresse geschrieben, zum zweiten ist der Eintrag WBENCHSCREEN in CUSTOMSCREEN geändert worden. Öffnen Sie ein zweites Fenster, das dann zum Beispiel das Label Screen_1 hat, müssen Sie vorher mit

```
move.l  Screen,Screen_1
```

auch dort die Screen-Adresse eintragen.

12.2 Fonts

Um nun zu zeigen (und um Sie auf den nächsten Absatz vorzubereiten), habe ich bei der Gelegenheit auch noch einen anderen Font eingesetzt. Vom Screen zeigt ein Zeiger auf »Font«. In dieser Struktur zeigt wieder ein Zeiger auf den Font-Namen. Probieren Sie ruhig andere Fonts und Größen (siehe Anhang) aus. Wichtig ist dieses Prinzip: »Zeiger zeigt auf Struktur, deren Zeiger zeigt auf andere Struktur«.

```
        opt     l-

*screen.s

        incdir  ":include/"
        include exec/exec_lib.i
        include intuition/intuition.i
        include intuition/intuition_lib.i
        include graphics/graphics_lib.i
        include graphics/text.i

        moveq   #0,d0                   ;Intuition oeffnen
        lea     int_name(pc),a1
        CALLEXEC OpenLibrary
        tst.l   d0
        beq     abbruch
        move.l  d0,_IntuitionBase

        moveq   #0,d0                   ;Graphics oeffnen
        lea     graf_name(pc),a1
        CALLEXEC OpenLibrary
        tst.l   d0
        beq     closeint
        move.l  d0,_GfxBase

        lea     NewScreen(pc),a0
        CALLINT OpenScreen              ;Open Screen
        tst.l   d0
        beq     closegfx
        move.l  d0,Screen               ;in Window eintragen!

        lea     NewWindow(pc),a0        ;Open Window
        CALLINT OpenWindow
        tst.l   d0
        beq     closescr
        move.l  d0,Window

        move.l  d0,a1                   ;Text ausgeben
        move.l  wd_RPort(a1),a1         ;Move
        moveq   #100,d0
```

```
        moveq   #50,d1
        CALLGRAF Move

        move.l  Window(pc),a0           ;print
        move.l  wd_RPort(a0),a1
        lea     msg(pc),a0
        moveq   #msglen,d0
        ALLGRAF Text

        move.l  Window(pc),a0           ;warte Event
        move.l  wd_UserPort(a0),a0
        move.b  MP_SIGBIT(a0),d1
        moveq   #0,d0
        bset    d1,d0
        CALLEXEC Wait

        move.l  Window(pc),a0           ;close all
        CALLINT CloseWindow

closescr
        move.l  Screen(pc),a0
        CALLINT CloseScreen

closegfx
        move.l  _GfxBase(pc),a1
        CALLEXEC CloseLibrary

closeint
        move.l  _IntuitionBase(pc),a1
        CALLEXEC CloseLibrary

abbruch
        rts


NewScreen       dc.w    0,0                     left, top
                dc.w    320,200                 width, height
                dc.w    2                       depth
                dc.b    0,1                     pens
                dc.w    0                       viewmodes
                dc.w    CUSTOMSCREEN            type
                dc.l    Font                    font
                dc.l    screen_title            title
                dc.l    0                       gadgets
                dc.l    0                       bitmap
Font
                dc.l    font_name
                dc.w    TOPAZ_SIXTY
                dc.b    FS_NORMAL
                dc.b    FPF_ROMFONT

W_Gadgets equ   WINDOWDRAG!WINDOWDEPTH!WINDOWCLOSE
```

```
W_Extras  equ     SMART_REFRESH!ACTIVATE
W_Title dc.b      'Fenster-Titel',Ø
        cnop      Ø,2

NewWindow
        dc.w      2Ø,2Ø                ;links, oben
        dc.w      3ØØ,1ØØ              ;Breite, Hoehe
        dc.b      Ø,1                  ;Pens des Screen
        dc.l      CLOSEWINDOW
        dc.l      W_Gadgets!W_Extras   ;Window Flags
        dc.l      Ø                    ;Kein User-Gadget
        dc.l      Ø                    ;keine User-Checkmark
        dc.l      W_Title              ;Titel des Window
Screen  ds.l      1                    ;eigener Screen
        dc.l      Ø                    ;keine Super Bitmap
        dc.w      1ØØ,2Ø,64Ø,2ØØ       ;Limits von Window-Groesse
        dc.w      CUSTOMSCREEN         ;Use our screen

_IntuitionBase  dc.l    Ø
_GfxBase        dc.l    Ø
Window          dc.l    Ø


int_name        INTNAME
graf_name       GRAFNAME
msg             dc.b    'Hallo Amiga'
msglen          equ     *-msg
                cnop    Ø,2
screen_title    dc.b    'Unser Screen',Ø
font_name       dc.b    'topaz.font',Ø
```

Bild 12.1: Eigener Screen und Text

12.3 Events

Mit Bild 12.2 komme ich zum Thema Diverses und damit zuerst zum Thema Events.

```
* event.s

* In diesen Files stecken diverse Deklarationen und Makros.
* Schauen Sie mal rein!

        incdir  ":include/"
```

```
        include intuition/intuition.i
        include intuition/intuition_lib.i
        include exec/exec_lib.i
        include graphics/graphics_lib.i

* Text im Fenster zeichnen
* ------------------------
PRINT   macro
        moveq   \1,dØ                   ;X-Position
        moveq   \2,d1                   ;Y
        move.l  windowptr,a1            ;Via Window-Zeiger
        move.l  wd_RPort(a1),a1         ;  Rast-Port-Adresse holen
        CALLGRAF Move                   ;Funktion Move to X,Y
        move.l  windowptr,a1            ;brauche wieder Rast-Port
        move.l  wd_RPort(a1),a1
        lea     \3,aØ                   ;Adresse Text
        moveq   \4,dØ               seine Laenge
        CALLGRAF Text                   ;und ausgeben
        endm

_main

* Intuition Library oeffnen:
* --------------------------
        lea     intname,a1
        moveq   #Ø,dØ
        CALLEXEC OpenLibrary
        tst.l   dØ
        beq     abbruch
        move.l  dØ,_IntuitionBase       ;Basis-Zeiger sichern

* Graphics Library oeffnen
* ------------------------
        lea    grafname,a1
        moveq  #Ø,dØ
        CALLEXEC OpenLibrary
        tst.l  dØ
        beq    closeint
        move.l dØ,_GfxBase              ;Basis-Zeiger sichern

* Window oeffnen
* --------------
        lea     windowdef,aØ            ;zeige auf Window-Struktur
        CALLINT OpenWindow              ;oeffne Window
        tst.l   dØ                      ;ging was schief?
        beq     closegraf               ;wenn ja
        move.l  dØ,windowptr            ;Window-Zeiger sichern

* Menu installieren
* -----------------
        move.l  dØ,aØ                   ;^Window
        lea     MenuØ,a1                ;^Menu
```

```
        CALLINT SetMenuStrip            ;do it

* Auf Event warten, dann auswerten
* -------------------------------

event
        move.l  windowptr,aØ            ;zeige auf Window-Struktur
        move.l  wd_UserPort(aØ),aØ      ;nun auf Message-Port
        move.l  aØ,a5                   ;rette Port-Adresse
        move.b  MP_SIGBIT(aØ),d1        ;Signal Bit holen
        moveq   #Ø,dØ                   ;Nummer in
        bset    d1,dØ                   ;Maske wandeln
        CALLEXEC Wait                   ;Schlaf gut!

        move.l   a5,aØ                  ;hole Port-Adresse
        CALLEXEC GetMsg                 ;hole Message
        move.l   dØ,a1                  ;muss nach a1
        move.l   im_Class(a1),d4        ;Msg-Typ
        move.w   im_Code(a1),d5         ;Untergruppe
        move.l   im_IAddress(a1),a4     ;Adr. f. Gadgets
        CALLEXEC ReplyMsg               ;quittiere Msg in a1

        cmpi.l   #CLOSEWINDOW,d4        ;Window Closed?
        beq      closewindow            ;wenn so

        cmpi.l   #MENUPICK,d4           ;Menu angewaehlt?
        beq      do_menu                ;wenn so

        cmpi.l   #GADGETUP,d4           ;und so fuer jedes
        beq      do_gadget              ;jedes Bit

        bra      event

do_menu
        cmpi     #MENUNULL,d5           ;Item selektiert?
        beq      event                  ;wenn nicht
        move     d5,d2                  ;Code
        lea      buffer+8,aØ            ;in hex anzeigen
        bsr      hex                    ;Routine aufrufen
        PRINT    #5Ø,#5Ø,buffer,#12
        bra      event

do_gadget
        sub.l    aØ,aØ
        CALLINT  DisplayBeep            ;Damit was passiert
        bra      event

closewindow
        move.l   windowptr,aØ           ;wir sind wieder wach
        CALLINT  CloseWindow            ;Fenster zu
```

```
* Libraries schliessen
* --------------------
closegraf
        move.l  _GfxBase,a1
        CALLEXEC CloseLibrary

closeint
        move.l  _IntuitionBase,a1
        CALLEXEC CloseLibrary

abbruch
        moveq   #Ø,dØ               ;oder normales Ende
        rts

* Konvertiere d2.w in ASCII-String ab (aØ)
* ----------------------------------------
hex
        moveq   #3,d1           ;fuer 4 Nibble
next    rol     #4,d2           ;hole 1 Nibble
        move    d2,d3           ;nach d3 retten
        and.b   #$Øf,d3         ;maskiere es
        add.b   #48,d3          ;in ASCII wandeln
        cmp.b   #58,d3          ;ist es >9 ?
        bcs     out             ;wenn nicht
        addq.b  #7,d3           ;sonst muss es A-F sein
out     move.b  d3,(aØ)+        ;1 Zeichen abspeichern
        dbra    d1,next         ;next nibble
        rts

buffer  dc.b    'Code =  xxxx  hex'
        cnop    Ø,2

W_Gadgets equ   WINDOWSIZING!WINDOWDRAG!WINDOWDEPTH!WINDOWCLOSE
W_Extras  equ   SMART_REFRESH!ACTIVATE

W_Title dc.b    'Fenster-Titel',Ø

windowdef
        dc.w    2ØØ,5Ø              ;links, oben
        dc.w    3ØØ,1ØØ             ;Breite, Hoehe
        dc.b    -1,-1               ;Pens des Screen
        dc.l    CLOSEWINDOW!MENUPICK!GADGETUP!GADGETDOWN
        dc.l    W_Gadgets!W_Extras  ;Window Flags
        dc.l    GadgetØ             ;erstes User-Gadget
        dc.l    Ø                   ;keine User-Checkmark
        dc.l    W_Title             ;Titel des Window
        dc.l    Ø                   ;kein eigener Screen
        dc.l    Ø                   ;keine Super Bitmap
        dc.w    1ØØ,2Ø              ;Min. Groesse
        dc.w    64Ø,2ØØ             ;Max.
        dc.w    WBENCHSCREEN        ;Use Workbench Screen
```

```
; Menue-Strukturen
; ----------------
MenuØ
        dc.l    Ø                   ;keines mehr
        dc.w    5Ø,Ø                ;x,y
        dc.w    6Ø,Ø                ;breit, hoch
        dc.w    MENUENABLED         ;Flags
        dc.l    MNameØ              ;^Titel
        dc.l    ItemØ               ;^Item-Liste
        dc.w    Ø,Ø,Ø,Ø             ;System-Use
ItemØ
        dc.l    Item1               ;^next Item
        dc.w    Ø,Ø                 ;x,y
        dc.w    1ØØ,12              ;breit, hoch
        dc.w    ITEMENABLED!ITEMTEXT!HIGHCOMP!COMMSEQ  ;Flags
        dc.l    Ø                   ;keine Excludes
        dc.l    INameØ              ;^Text
        dc.l    Ø                   ;Select Fill
        dc.b    'N'                 ;Cmd-Key
        dc.b    Ø                   ;Dummy
        dc.l    Ø                   ;Kein Sub-Item
        dc.w    Ø                   ;     next Select

Item1
        dc.l    Ø                   ;^next Item
        dc.w    Ø,12                ;x,y
        dc.w    1ØØ,12              ;breit, hoch
        dc.w    ITEMENABLED!ITEMTEXT!HIGHCOMP!COMMSEQ  ;Flags
        dc.l    Ø                   ;keine Excludes
        dc.l    IName1              ;^Text
        dc.l    Ø                   ;Select Fill
        dc.b    'E'                 ;Cmd-Key
        dc.b    Ø                   ;Dummy
        dc.l    Ø                   ;Kein Sub-Item
        dc.w    Ø                   ;     next Select

INameØ
        dc.b    Ø,2                 ;Pens
        dc.b    RP_JAM1,Ø           ;Schreibmodus
        dc.w    2,2                 ;x,y
        dc.l    Ø                   ;System-Font
        dc.l    striØ               ;^Text
        dc.l    Ø                   ;kein Text mehr
IName1
        dc.b    Ø,2                 ;Pens
        dc.b    RP_JAM1,Ø           ;Schreibmodus
        dc.w    2,2                 ;x,y
        dc.l    Ø                   ;System-Font
        dc.l    stri1               ;^Text
        dc.l    Ø                   ;kein Text mehr

MNameØ  dc.b    'Menu Ø',Ø
```

```
        cnop    Ø,2
striØ   dc.b    'Item Ø',Ø
        cnop    Ø,2
stri1   dc.b    'Item 1',Ø
        cnop    Ø,2

* Gadget-Strukturen
* -----------------

GadgetØ
        dc.l    Gadget1         ;noch eins
        dc.w    2Ø,2Ø           ;x,y
        dc.w    4Ø,2Ø           ;breit,hoch
        dc.w    GADGHCOMP       ;Flags
        dc.w    RELVERIFY       ;Aktivierung
        dc.w    BOOLGADGET      ;Typ
        dc.l    BorderØ         ;^Border-Strukt.
        dc.l    Ø
        dc.l    GØtext          ;^Text-Strukt.
        dc.l    Ø,Ø
        dc.w    Ø               ;ID
        dc.l    Ø
BorderØ
        dc.w    Ø,Ø             ;x,y
        dc.b    15,Ø            ;Pens
        dc.b    RP_JAM1
        dc.b    5               ;Paare
        dc.l    paarØ           ;^Liste
        dc.l    Ø               ;kein Border mehr
paarØ
        dc.w    Ø,Ø             ;Koord.-Liste
        dc.w    Ø,19
        dc.w    39,19
        dc.w    39,Ø
        dc.w    Ø,Ø
GØtext
        dc.b    7,Ø             ;Pens
        dc.b    RP_JAM1
        dc.w    1Ø,7            ;x,y
        dc.l    Ø               ;Sys-Font
        dc.l    strgØ
        dc.l    Ø
strgØ   dc.b    'GØ',Ø
        cnop    Ø,2

Gadget1
        dc.l    Ø
        dc.w    8Ø,2Ø           ;x,y
        dc.w    4Ø,2Ø           ;breit,hoch
        dc.w    GADGHCOMP       ;Flags
        dc.w    RELVERIFY       ;Aktivierung
        dc.w    BOOLGADGET      ;Typ
```

```
        dc.l     Border1          ;^Border-Strukt.
        dc.l     Ø
        dc.l     G1text           ;^Text-Strukt.
        dc.l     Ø,Ø
        dc.w     Ø                ;ID
        dc.l     Ø
Border1
        dc.w     Ø,Ø              ;x,y
        dc.b     15,Ø             ;Pens
        dc.b     RP_JAM1
        dc.b     5                ;Paare
        dc.l     paar1            ;^Liste
        dc.l     Ø                ;kein Border mehr
paar1
        dc.w     Ø,Ø              ;Koord.-Liste
        dc.w     Ø,19
        dc.w     39,19
        dc.w     39,Ø
        dc.w     Ø,Ø
G1text
        dc.b     7,Ø              ;Pens
        dc.b     RP_JAM1
        dc.w     1Ø,7             ;x,y
        dc.l     Ø                ;Sys-Font
        dc.l     strg1
        dc.l     Ø
strg1   dc.b     'G1',Ø
        cnop     Ø,2

intname          INTNAME          ;Name Intuition Lib (via Makro)
grafname         GRAFNAME         ;Name Graphics Lib

_IntuitionBase   ds.l    1        ;Speicher fuer Zeiger
_GfxBase         ds.l    1
windowptr        ds.l    1
```

Bild 12.2: Das Event-Handling in Intuition

Ein Event ist ein Ereignis, und aus Sicht des Computers sind die User-Eingaben, die da alle Ewigkeit einmal auftreten, tatsächlich ein Ereignis. Auf jeden Fall sieht es so aus: Sie kennen schon die Sequenz

```
        move.l   windowptr,aØ          ;zeige auf Window-Struktur
        move.l   wd_UserPort(aØ),aØ    ;nun auf Message-Port
        move.l   aØ,a5                 ;rette Port-Adresse
        move.b   MP_SIGBIT(aØ),d1      ;Signal Bit holen
        moveq    #Ø,dØ                 ;Nummer in
        bset     d1,dØ                 ;Maske wandeln
        CALLEXEC Wait                  ;Schlaf gut!
```

An dieser Stelle geht unser Task schlafen. Geweckt wird er erst, wenn ein Event auftritt. Was dann sein kann, haben wir in der Fensterdefinition bestimmt. Bisher haben wir immer unterstellen können, daß das Event CLOSEWINDOW hieß. Doch nun habe ich folgende Bits per Oder-Verknüpfung (! in Assembler) als Events zugelassen:

```
dc.l    CLOSEWINDOW!MENUPICK!GADGETUP!GADGETDOWN
```

Da muß ich dann wohl oder übel prüfen, welches Event von diesen denn nun eingetreten ist. Dazu muß ich die Message (Nachricht) lesen, und das geht so:

```
move.l   a5,aØ                ;hole Port-Adresse
CALLEXEC GetMsg               ;hole Message
move.l   dØ,a1                ;muss nach a1
```

A1 zeigt nun auf den Message-Port, das ist auch (wieder) eine Struktur. Wie üblich können wir nun wieder über Offsets auf die Elemente dieser Struktur zugreifen, wobei folgende interessieren:

```
move.l   im_Class(a1),d4      ;Msg-Typ
move.w   im_Code(a1),d5       ;Untergruppe
move.l   im_IAddress(a1),a4   ;Adr. f. Gadgets
```

Der Typ entspricht genau unseren Flag-Bits des Windows. Man kann also zum Beispiel schreiben:

```
cmpi.l   #CLOSEWINDOW,d4      ;Window Closed?
beq      closewindow          ;wenn so
```

Der Code gibt eine Untergruppe an. Wurde zum Beispiel der Typ MENUPICK erkannt, kann ich in Code nachsehen, welches Menü und welches Untermenü das war.

Die Adresse, die ich hier in a4 gerettet habe, braucht man im Fall von Gadgets. Diese Adresse zeigt auf den Beginn der Gadget-Struktur (deren Gadget angeklickt wurde). Wieder über ein Offset kann ich da die ID (die Nummer des Gadgets) so holen:

```
move    gg_GadgetID(a4),dØ
```

Was auch immer: Bevor wir Intuition oder eine andere System-Routine aufrufen, müssen wir unbedingt die Nachricht quittieren. Hier geschieht das mit

```
CALLEXEC ReplyMsg             ;quittiere Msg in a1
```

Lassen Sie bis zum »Reply« auch sonst nicht zuviel Zeit vergehen. Hier liegen vier Befehle dazwischen, das geht noch.

12.4 Menüs

Bisher hatten wir zwar mit MENUPICK zugelassen, im Fall von Menü-Auswahlen geweckt zu werden, doch vorher ist einiges zu tun. Die Menüs werden installiert mit

```
move.l  dØ,aØ                    ;^Window
lea     MenuØ,a1                 ;^Menu
CALLINT SetMenuStrip             ;do it
```

Beachten Sie, daß ein Menü immer an ein Fenster gebunden ist (obwohl es immer in der Schirmzeile erscheint). Daher der Zeiger auf die Fensterstruktur als erster Parameter. Dann müssen wir einen Zeiger auf die Menü-Struktur übergeben und können schließlich den »Menü-Streifen« installieren.

Die Arbeit liegt in der Strukturerstellung. Das ist viel Tipperei aber sonst ganz einfach, zumal man eigentlich nur ein Menü schreibt und den Rest dupliziert und nur an wenigen Stellen ändert. Ich meine zwar, Sie haben das Prinzip schon erkannt, aber für die anderen Leser:

Es gibt für jeden Menü-Titel eine Struktur, wobei immer die eine auf die nächste zeigt, bis der Zeiger der letzten null ist. In jeder Titelstruktur gibt es einen Zeiger auf die Item-Liste (Untertitel), die dem gleichen Prinzip folgt. In jeder Item-Struktur gibt es einen Zeiger auf eine Textstruktur und in dieser wieder einen Zeiger auf den Text an sich. Genug gezeigert? Ich meine, machen wir eine Pause, wir zeigern gleich weiter.

In der Routine do_menu zeige ich den Code einfach als Hex-Wort an. Die zwei Menüs reichen nicht zur Auflösung des Rätsels, daher hier die Lösung. Im 16-Bit-Wort von Code stecken drei Zahlen, nämlich die Nummern für den Titel, das Item und das Sub-Item. Für letztere werden in den Items Zeiger auf Strukturen eingetragen, die sich wie Items verhalten. Der Code löst sich auf in

Bit 0- 4: Titel
Bit 5-10: Item
Bit 11-15: Sub-Item

Die Nummern zählen immer ab 0. Typisch wird man das Register kopieren, für den Titel das Wort mit #%11111 »unden« und hat so die Titelnummer. In der Titelroutine nimmt man wieder den Original-Code, schiebt den um 5 Bit nach rechts, und »undet« mit %#111111 und hat das Item. Waren bei bestimmten Items noch Sub-Items vorgesehen, geht es entsprechend weiter (um 11 schieben und mit #%11111 unden). Beachten Sie dazu, daß man mittels einer Konstanten nur maximal 8 schieben kann, für 11 wären dann zwei Befehle (mit 8 und mit 3) erforderlich. Alternativ lädt man ein Register mit 11 und schiebt damit.

12.5 Gadgets

Wie man eine Gadget-ID erhält, hatte ich schon geschildert, zwei Voraussetzungen fehlen allerdings noch. Im Window muß »User-Gadget« auf die erste Gadget-Struktur zeigen, und diese muß es natürlich geben.

Hier geht nun wieder die »Zeigerei« los. Ein Gadget zeigt auf das nächste. In einem Gadget gibt es einen Zeiger auf eine Border-Struktur (Vieleck, sonst Polygon geheißen). Dort gibt es wieder einen Zeiger auf eine Liste mit den Koordinatenpaaren für jede Ecke des Polygons. Wir bescheiden uns hier mit einem schlichten Quadrat. Ansonsten gibt es noch einen Zeiger auf die schon bekannte Textstruktur, von der dann wieder ein Zeiger auf den Textstring zeigt.

12.6 Requester

Ich könnte nun viele Menü-Punkte und Gadgets erzeugen, damit schön Seiten füllen, Sie aber langweilen. Auch andere Strukturen sind nicht schwieriger. Genau steht alles im »Intuition Reference Manual«, dessen Lektüre ich Ihnen sehr empfehlen kann. Zur Übung sollten Sie aber einmal einen Requester in das Programm einbauen, dessen Aufruf zum Beispiel an der Stelle stehen kann, an der ich jetzt mit »DisplayBeep« den Schirm kurz rot blitzen lasse.

Der einfachste Fall ist der Auto-Requester, der im Prinzip eine Frage stellt und zwei Antworten (zwei Gadgets) zuläßt. Der Aufruf erfolgt mit

```
        CALLINT   AutoRequest
```

wobei vorher geladen werden sollten:

a0:	Zeiger auf Fensterstruktur (hier lea windowptr,a0)
a1:	Zeiger auf Textstruktur mit der Frage (Nehmen Sie doch einfach lea MName0,a1)
a2:	Zeiger auf Textstruktur mit Text, linkes Gadget
a3:	Zeiger auf Textstruktur mit Text, rechtes Gadget (Nehmen Sie doch einfach lea G0text,a2 lea G1text,a3)
d0,d1:	IDCMP-Flags, die angeben, was wie Klicken in das linke/rechte Gadget erkannt werden soll. (Nehmen Sie vorerst clr.l d0 / clr.l d1)
d2.l,d3.l:	Breite und Höhe des Requesters

12.7 C in Assembler umschreiben

Im Hardware-Manual finden Sie schöne Beispiele für die direkte Programmierung der Grafik-Hardware in Assembler. Wenn Sie extrem schnelle Grafik interessiert, empfehle ich den Kauf dieses Werkes. Ansonsten sind die meisten Beispiele der Amiga-Dokumentation in C geschrieben, auch Intuition, was Grund genug ist, uns einmal um die Übersetzung in Assembler zu kümmern.

Ein »#include« ersetzen Sie durch »include«, beim File-Namen ändern Sie die Extension von h in i. Für

```
#define Name Wert
```

schreiben Sie

```
Name equ  Wert
```

Die Namen der Funktionen stimmen überein, nur daß wir immer _LVO davorsetzen müssen. Nehmen wir allerdings die Makros, kommen wir C noch näher, denn zum Beispiel

```
move.l   _IntuitionBase,a6
jsr      _LVOOpenWindow(a6)
```

heißt mittels Makro nur

```
CALLINT   OpenWindow
```

Die Datentypen in C machen viel Lärm um wenig, wobei gilt:

C-Typ	Assembler-Typ
int	.W
long int	.L
unsigned int	.W
char	.B
BYTE	.B
UBYTE	.B
WORD	.W
UWORD	.W
LONG	.L
ULONG	.L

Die groß geschriebenen Typen sind übrigens keine echten C-Typen, sondern nur Makros. Das in C beim Aufruf übliche

```
if OpenWindow(&windowptr)==Ø
    exit(false)
```

ersetzen Sie durch

```
tst.l  dØ
beq    exit  ;dort Close Libs und so und rts
```

Ein Ausdruck wie

```
windowprt->wd_RPort
```

kostet uns in Assembler die Zeilen

```
move.l  windowptr,a1
move.l  wd_RPort(a1),a1
```

Die Strukturen werden Sie schon anhand der Namen wiedererkennen. Manchmal finden Sie auch nur deren Namen mit dem Präfix »extern«. Das ist dann eine Anweisung an den Linker, diese Struktur einzubinden.

Zum Schluß (und Trost): Assembler ist immer schneller, nicht nur weil der Code, den Sie schreiben, besser ist (maßgeschneiderter) als der des C-Compilers, sondern auch aus einem simplen Grund. Die Parameterübergabe an die System-Routinen läuft immer über Register, wie Sie es bisher auch praktiziert haben. In C läuft das aber per Sprachdefinition über den Stack. Folglich muß der C-Compiler einen Code erzeugen, der erst die Parameter auf den Stack packt und dann immer ein sogenanntes Binding erzeugt, das diese Parameter wieder vom Stack holt und in die Register lädt.

Kapitel 13

Einbindung von Assembler-Routinen in BASIC

Anforderungen an die Routinen

Raum für Routinen

Laden und Aufrufen

Die Parameterübergabe

CLI-Befehle in BASIC aufrufen

13.1 Anforderungen an die Routinen

Lageunabhängig

Im Amiga gibt es kein sicheres Plätzchen, wie zum Beispiel das Stück nicht genutzten Video-RAM des Atari ST. Ansonsten ist diese Technik auch nicht zu empfehlen, denn sollte wirklich jemand einen unbelegten Speicherbereich beim Amiga entdecken, so werden alle ihre Assembler-Routinen da speichern wollen, was dann zu Konflikten führt.

Daraus folgt die erste Forderung an die Routine, sie muß lageunabhängig sein, neudeutsch: position independend. Damit sind leider Adressierungen wie

```
move.l    #buffer,d2
```

nicht erlaubt. Abhilfe bringt die PC-relative Adressierung (siehe auch Kapitel 3.6). Nun wäre eine Schreibweise wie »move.l buffer(pc),d2« natürlich Unsinn, denn »buffer« ist eine Adresse und keine Adreßdistanz. Die könnte man errechnen, einfacher ist jedoch diese Form:

```
lea      buffer(pc),a0
move.l   a0,d2
```

Bliebe noch ein Problem: Die PC-relative Adressierung ist beim Zieloperanden nicht möglich. Verboten ist also zum Beispiel

```
move     #1,dahin(pc)     ;falsch!
```

Auch hier bringen zwei Befehle Abhilfe, nämlich

```
lea      dahin(pc),a0
move     #1,(a0)
```

Um zum Beispiel in eine Window-Struktur zu schreiben, kann man aber auch so vorgehen:

```
lea      windowdef(pc),a0
move     #1,4(a0)
```

Das wäre der richtige Ersatz für

```
move     #1,windowdef+4
```

Nur ein Segment

Sie dürfen für Programm-, Daten- und Variablenspeicher nur ein Segment bilden. Das ist kein Problem, Sie müssen nur Anweisungen wie SECTION, DATA und BSS weglassen. Der Grund ist, daß die PC-relative Adressierung nur eine vorzeichenbehaftete 16-Bit-Distanz erlaubt (+/- 32 Kbyte), der Lader aber die Segmente auf größere Distanz legen kann. Logisch, daß auch Ihr Programm nicht größer als 32 Kbyte werden darf, zumindest dürfen Befehle und zugehörige Adressen nicht um mehr als 32 Kbyte auseinanderliegen.

Unterprogramm muß alle Register retten

Daß die Routine mit RTS enden muß, ist klar. Schließlich soll sie auch zu BASIC zurückkehren. Wichtig ist, daß Sie zu Beginn der Routine alle Register retten (movem) und vor dem RTS wieder zurückspeichern.

Nur Code und Daten speichern

Ihre Assembler-Routine sollten Sie nur assemblieren, nicht jedoch linken. Letzteres fügt noch diverse Infos für den Lader (des DOS) hinzu, die Sie in BASIC nicht brauchen. Kann Ihr Assembler keine reinen Objekt-Module abspeichern, müssen Sie den Overhead selbst entfernen. Ein Beispiel dafür kommt noch.

13.2 Raum für Routinen

Den Raum für die Routine muß das BASIC-Programm bereitstellen. Dazu kann man in BASIC einen entsprechend großen Array definieren oder auch einfach einen String verwenden, der von selbst groß genug wird, wenn man den Code da hinein lädt (wird noch gezeigt, wie).

13.3 Laden und Aufrufen von Assembler-Routinen

Mit Bild 13.1 soll die Assembler-Routine vorgestellt werden, mit der wir uns zuerst befassen wollen. Das Programm öffnet ein Fenster, wartet auf eine Taste und ist dann auch schon fertig. Hier geht es mir in erster Linie um die Darstellung, wie man auf Datenbereiche lageunabhängig zugreift.

```
        opt     l+,p+   ;linkbar, pos. indep.
* b1.s
        include dos_equates             ;_LVO-Equates einfügen!!!

RELO    macro
        lea     \1(pc),aØ
        move.l  aØ,\2
        endm
_main
        movem.l dØ-d6/aØ-a6,-(sp)       ;fuer Basic retten
        lea     dosname(pc),a1          ;Name der DOS-Lib
        moveq   #Ø,dØ                   ;Version egal
        move.l  _SysBase,a6             ;Basis Exec
        jsr     _LVOOpenLibrary(a6)     ;DOS-Lib oeffnen
        tst.l   dØ                      ;Fehler?
        beq     fini                    ;wenn Fehler, Ende
        move.l  dØ,a6                   ;Zeiger merken

        RELO    name,d1
        move.l  #1ØØ5,d2                ;Status = gibt es
        jsr     _LVOOpen(a6)            ;nun oeffnen
        move.l  dØ,d5                   ;Handle merken
        tst.l   dØ                      ;Fehler?
        beq     fini                    ;wenn ja, abbrechen

        move.l  d5,d1                   ;von CON lesen
        RELO    buffer,d2               ;in diesen Puffer
        move.l  #1,d3                   ;1 Zeichen
        jsr     _LVORead(a6)            ;Lesen aufrufen

        move.l  d5,d1                   ;CON wieder schliessen
        jsr     _LVOClose(a6)

        move.l  a6,a1                   ;DOS-Lib-Basis
        move.l  _SysBase,a6             ;Basis Exec
        jsr     _LVOCloseLibrary(a6)    ;Funktion "Schliessen"
        movem.l (sp)+,dØ-d6/aØ-a6
fini    rts                             ;Return zum CLI

dosname dc.b    'dos.library',Ø
        cnop    Ø,2
name    dc.b    'CON:4Ø/1ØØ/58Ø/8Ø/hit any key',Ø
        cnop    Ø,2
buffer  ds.b    2
```

Bild 13.1: Programm 1, das von BASIC her aufgerufen werden soll

Das »position independend« habe ich mit dem Makro RELO etwas rationalisiert. Der Hinweis kommt zwar spät, aber immerhin noch: Auch in Programmen, die nicht von BASIC her aufgerufen werden sollen, bringt die Methode Vorteile. Dadurch entfallen nämlich im Code die Relokatier-Offsets, womit der Code kürzer und damit schneller geladen wird. Auch wenn Sie auf RELO verzichten und nur da, wo es eh möglich ist, ein »(pc)« einsetzen, bringt das schon etwas. Der Compiler-Switch »p+« (DEVPAC) ist übrigens ganz nützlich. Er erzeugt zwar keinen lageunabhängigen Code, meldet aber die lageabhängigen Zeilen als Fehler.

Wenn ein BASIC-Programm eine Assembler-Routine aufrufen soll, muß diese natürlich im RAM stehen. Die einfachste Methode ist, den Binär-File mit dem Code dorthin zu laden. Doch leider ist das für den Anwender ziemlich lästig, da er außer dem BASIC-Programm auch immer den zugehörigen Code-File parat haben muß. Trotzdem soll das Verfahren vorgestellt werden, denn es ist (zum Testen) so schön einfach und schnell. Bild 13.2 zeigt die Lösung.

```
OPEN":buch/a" AS 1
l=LOF(1)
CLOSE 1

OPEN ":buch/a" AS 1 LEN=l
FIELD #1, l AS a$
GET 1,1
ass$=a$
CLOSE 1

ass&=SADD(ass$)
CALL ass&

END
```

Bild 13.2: Methode 1: BASIC und Assembler getrennt

Mein Code-File hat den schlichten Namen a und steckt im Directory Buch. Mittels der LOF-Funktion wird geprüft, wie lang der File ist, das Ergebnis wird in der Variablen l notiert.

Nun wird der File nochmals geöffnet, jetzt aber als Random-File. Die Recordlänge wird gleich der File-Größe gesetzt, womit nur ein »get« ausreicht, den File im Stück einzulesen (das ist wichtig). Damit steht der Code im String ass$. Mit SADD kann man nun die Adresse des Strings feststellen und der Variablen ass& zuweisen. Beachten Sie

den &-Operator! Damit wird der Typ Long-Integer erzwungen, was für eine Adresse Grundvoraussetzung ist. Nun kann man schlicht mit

```
CALL Adreßvariable
```

das Maschinenprogramm aufrufen. Wie gesagt, die Methode hat den Nachteil, daß man immer beide Module, den Code und das BASIC-Programm auf der Disk haben muß, eine Tatsache, die ein reiner Anwender (beim Kopieren) leider nur zu oft vergißt. Abhilfe bringt die Lösung nach Bild 13.3.

```
DIM  a%(72)

FOR i=1 TO 68
  READ a$:a%(i)=VAL("&h"+a$)
NEXT

ass&=VARPTR(a%(1))
CALL ass&
END

DATA 48E7,FEFE,43FA,0058,7000,2C79,0000,0004
DATA 4EAE,FDD8,4A80,6700,0044,2C40,41FA,004C
DATA 2208,243C,0000,03ED,4EAE,FFE2,2A00,4A80
DATA 6700,002A,2205,41FA,0050,2408,263C,0000
DATA 0001,4EAE,FFD6,2205,4EAE,FFDC,224E,2C79
DATA 0000,0004,4EAE,FE62,4CDF,7F7F,4E75,646F
DATA 732E,6C69,6272,6172,7900,434F,4E3A,3430
DATA 2F31,3030,2F35,3830,2F38,302F,6869,7420
DATA 616E,7920,6B65,7900
```

Bild 13.3: Methode 2: BASIC und Assembler in einem File

Das Maschinenprogramm steht als Hex-Strings in DATA-Zeilen. Mit der einfachen Anweisungsfolge in der FOR-Schleife wird es in einen Array geschrieben. Dieser Array muß (!!) vom Typ Integer sein, ansonsten würde nämlich BASIC die Zahlen in das Fließkomma-Format bringen, womit von unserem Code nichts übrig bliebe.

Das erste Wort des Codes steht nun in a%(1). Mit VARPTR kann man dessen Adresse ermitteln. Den Rest kennen Sie schon. Doch wie kommt der Code in die Data-Zeilen? Die kleine Utility, von Bild 13.4 löst genau dieses Problem, natürlich in Assembler, um den es hier geht (BASIC wäre einfacher, aber langweilig).

```
        opt      l-                              ;nur wenn DEVPAC-Assembler

* ABC = Assembler Basic Converter
* Object-File -> Basic-DATA-Zeilen
* (c) 1987 Peter Wollschlaeger
* --------------------------------

_SysBase         equ     4                       ;Basis von Exec
_LVOOpenLibrary  equu    -552                    ;Library oeffnen
_LVOCloseLibrary equ     -414                    ;Library schliessen

_LVOOutput       equ     -60                     ;StdOut-Handle holen
_LVOWrite        equ     -48                     ;Schreiben (auf File)
_LVORead         equ     -42                     ;Lesen
_LVOOpen         equ      -30                    ;File oeffnen
_LVOClose        equ      -36                    ;     schliessen
MODE_OLDFILE     equ     1005                    ;File existiert
MODE_NEWFILE     equ     1006                    ;File neu/ueberschreiben

* Einige simple Makros zum File-Handling
* --------------------------------------

OPEN    macro
        move.l  \1,d1           ;Name
        move.l  \2,d2           ;Mode
        jsr     _LVOOpen(a6)
        endm

READ    macro
        move.l  \1,d1           ;Handle
        move.l  \2,d2           ;Puffer-Adr
        move.l  \3,d3           ;Max.
        jsr     _LVORead(a6)
        endm

WRITE   macro
        move.l  \1,d1           ;Handle
        move.l  \2,d2           ;Puffer-Adr
        move.l  \3,d3           ;Anzahl
        jsr     _LVOWrite(a6)
        endm

CLOSE   macro
        move.l  \1,d1           ;Handle
        jsr     _LVOClose(a6)
        endm
* --------------------------------------

        movem.l  a0/d0,-(sp)                     ;Parms. Kommandozeile
```

```
* DOS-Lib oeffnen:
_main   move.l  #dosname,a1             ;Name der DOS-Lib
        moveq   #0,d0                   ;Version egal
        move.l  _SysBase,a6             ;Basis Exec
        jsr     _LVOOpenLibrary(a6)     ;DOS-Lib oeffnen
        tst.l   d0                      ;Fehler?
        beq     fini                    ;wenn Fehler, Ende
        move.l  d0,a6                   ;Zeiger merken

* Kommandozeile parsen
* --------------------
        movem.l (sp)+,a0/d0             ;hole Parms
        move.b  #' ',-1(a0,d0.l)        ;terminiere mit Blank weil
;                                         mein Parser das will
        lea     quelle,a1               ;Puffer Name Quellfile
        bsr     parse                   ;hole ihn
        lea     ziel,a1                 ;Name Zielfile
        bsr     parse                   ;dito

* Beide Files oeffnen
* -------------------
        OPEN    #quelle,#MODE_OLDFILE
        move.l  d0,d4                   ;Handle
        tst.l   d0                      ;ging was schief?
        bne     weiter                  ;wenn nicht
        jsr     _LVOOutput(a6)          ;Hole Output-Handle
        WRITE   d0,#msg1,#len1          ;und melde Fehler
        bra     cl_lib                  ;und Ende

weiter  OPEN    #ziel,#MODE_NEWFILE     ;wie vor nun mit Zielfile
        move.l  d0,d5
        tst.l   d0
        bne     start
        jsr     _LVOOutput(a6)          ;Hole Output-Handle
        WRITE   d0,#msg2,#len2
        bra     cl_q

* Hier geht's nun los
* -------------------
start   READ    d4,#buffer,#16          ;Header skippen

loop    READ    d4,#buffer,#16          ;Lese 16 Bytes oder weniger
        tst.l   d0                      ;End of File?
        beq     cl_z                    ;wenn ja

        asr     #1,d0                   ;gelesene Bytes durch 2
        subq    #1,d0                   ;- 1 wegen dbra-loop
        move    d0,d6                   ;Anzahl Worte merken

        WRITE   d5,#data,#6             ;write LF + 'DATA '

        lea     buffer,a4               ;Zeiger auf Quelle
```

```
conv    move    (a4)+,d2                ;ein Wort
        lea     hbuf,aØ                 ;        soll dahin
        bsr     hex                     ;als Hex-String
        WRITE   d5,#hbuf,#4             ;auf Zielfile
        cmp     #Ø,d6                   ;letztes Wort in Zeile?
        beq     no                      ;wenn nicht
        WRITE   d5,#komma,#1            ;sonst ein Komma dahinter
no      dbra    d6,conv                 ;bis Zeile fertig
        bra     loop                    ;bis EOF

cl_z    CLOSE   d5                      ;Close Zielfile
cl_q    CLOSE   d4                      ;und Quellfile

cl_lib  move.l  a6,a1                   ;DOS-Lib-Basis
        move.l  _SysBase,a6             ;Basis Exec
        jsr     _LVOCloseLibrary(a6)    ;Funktion "Schliessen"

fini    rts                             ;Return zum CLI

* Einfacher Parser; kennt nur Blanks als Delimeter
* ---------------------------------------------
parse   cmp.b   #' ',(aØ)+              ;fuehrende Blanks
        beq     parse                   ;skippen
        subq.l  #1,aØ                   ;wir waren 1 zu weit
p1      move.b  (aØ)+,(a1)+             ;nun kopiere
        cmp.b   #' ',(aØ)               ;bis wieder ein Blank
        bne     p1
        clr.b   (a1)                    ;Terminiere mit Ø-Byte
        rts

* Konvertiere d2.w in ASCII-String ab (aØ)
* ----------------------------------------
hex
        moveq   #3,d1           ;fuer 4 Nibble
next    rol     #4,d2           ;hole 1 Nibble
        move    d2,d3           ;nach d3 retten
        and.b   #$Øf,d3         ;maskiere es
        add.b   #48,d3          ;in ASCII wandeln
        cmp.b   #58,d3          ;ist es >9 ?
        bcs     out             ;wenn nicht
        addq.b  #7,d3           ;sonst muss es A-F sein
out     move.b  d3,(aØ)+        ;1 Zeichen abspeichern
        dbra    d1,next         ;next nibble
        rts

* ------------------------------------------------------
* Datenbereich:

dosname dc.b    'dos.library',Ø
        cnop    Ø,2
```

```
data     dc.b   1Ø,'DATA '
         cnop   Ø,2
komma    dc.b   ','
         cnop   Ø,2
msg1     dc.b   'Quellfile nicht gefunden',1Ø
len1     equ    *-msg1
         cnop   Ø,2
msg2     dc.b   'Konnte Zielfile nicht oeffnen',1Ø
len2     equ    *-msg2
         cnop   Ø,2

quelle   ds.b   4Ø
ziel     ds.b   4Ø
buffer   ds.b   16
hbuf     ds.b   4
```

Bild 13.4: Assembler-Programm generiert BASIC-Zeilen

Das Listing mitsamt Beschreibung habe ich schon einmal im 68000er veröffentlicht, damals als Einführung für 68000er-Profis in die Amiga-Programmierung. Da Sie schon weiter sind, streiche ich den größten Teil des Artikeltextes. Einiges lasse ich sozusagen als Repetitorium stehen. Das Programm namens ABC wird mit der Syntax

```
ABC Object_File Basic_File
```

im CLI aufgerufen. Object_File ist das Maschinenprogramm, so wie es der Assembler erzeugt. BASIC_File ist danach reiner ASCII-Text, in dem jede Zeile mit dem Wort DATA beginnt. In den DATA-Zeilen steht das Maschinenprogramm in Form von Strings. Jeder String ist ein Wort in hexadezimaler Notation. Die hexadezimale Schreibweise hat den Vorteil, daß man ein Programm besser lesen kann.

Zwischen mir und dem Amiga wurde vereinbart, daß in jeder Zeile acht Wörter stehen sollen. Daraus folgt dann folgendes Schema für das Programm:

1. Zerlege Kommandozeile in die Filenamen Quelle und Ziel
2. Öffne die Files
3. Schreibe ein Linefeed plus das Wort DATA plus 1 Blank
4. Lese 16 Byte
3. Wandle 16 Byte in acht Hexstrings
6. Schreibe acht Strings durch Kommas getrennt
7. Wiederhole ab 3. bis EOF(Quelle)
8. Schließe die Files

Kleine Schwierigkeit: Es bleiben zum Schluß nicht 16 Byte übrig, folglich gilt: Lese 16 Byte oder was noch da ist. Wandle davon die Hälfte in Hex-Worte. Natürlich muß man

auch prüfen, ob beim Öffnen der Files etwas schiefging und gegebenenfalls eine Fehlermeldung ausgeben.

Damit hätten wir die Aufgabe beschrieben, nun zur Lösung. Nach dem Programmstart steht die Adresse der Kommandozeile (aller Text, der dem Programmnamen folgt) im Register A0, ihre Länge in D0. Es ist üblich, zuerst diese Parameter zu retten, dann zuzusehen, ob man seine Libs öffnen kann, und danach die Kommandozeile zu interpretieren. Sie können natürlich die beiden »movem« sparen, indem Sie das Öffnen der DOS-Lib nach dem »parsen« schreiben.

Wie auch immer, der Parser kopiert die beiden File-Namen von der Kommandozeile in zwei Puffer und schließt sie (DOS will es so) mit einem Null-Byte ab. Nun versuche ich, die beiden Files zu öffnen. Im Fehlerfall hole ich mittels »_LVOOutput« die Handle der Standard-Ausgabe (hier das CLI-Fenster) und gebe mit demselben WRITE-Makro die Fehlermeldung aus.

Hier liegt übrigens der Grund dafür, daß Sie das Programm nur vom CLI aus aufrufen dürfen. Auch »RUN ABC...« ist nicht erlaubt. Wenn Sie das wollen, müssen Sie ein eigenes Fenster öffnen und das geht so:

```
        move.l  #window,d1              ;Adresse
        move.l  #MODE_OLDFILE,d2        ;Status = gibt es
        jsr     _LVOOpen(a6)            ;nun oeffnen
```

Mit unserem Makro wird die Sache noch einfacher, also

```
        OPEN #window,#MODE_OLDFILE
```

Danach steht die Handle in D0 und kann sowohl für die Eingabe (Read) als auch die Ausgabe (Write) eingesetzt werden. Im Datenbereich fehlt dann noch

```
window   dc.b   'CON:40/100/580/80/Fenstertitel',0
```

Die vier Zahlen beschreiben die linke obere Ecke des Fensters, seine Breite und Höhe. Danach folgt der Titel. Nach dem Aufruf sollten Sie die Handle (D0) sichern. Vergessen Sie nicht, das Fenster wieder zu schließen.

Nun zeige mir jemand einen 68000er, bei dem sich Windows so einfach öffnen lassen (mittels System-Software)! Dieses Fenster hat zwar nicht die Mächtigkeit eines Intuition-Windows, ist dafür aber sehr bequem zu handhaben.

Ab »Hier geht's nun los« werden zuerst 16 Byte blind gelesen (kommt gleich), danach läuft die schon geschilderte Read/Write-Schleife. Zum Schluß werden die Files geschlossen und schließlich noch die DOS-Library.

Die 16 Byte, die ich anfangs überlese, sind der sogenannte File-Header. Sie finden da zuerst ein Langwort des Inhalts $000003E7. Das heißt, hier beginnt eine Programm-Unit. Danach folgt ein Langwort des Inhalts 0, sofern Sie dem Modul keinen Namen gegeben haben (sollten Sie auch nicht tun). Das letzte Langwort könnte der Grund sein, warum Sie vielleicht die Zeile ändern und nur 12 Byte überlesen wollen. Hier steht nämlich die Länge des Moduls in Langworten minus eins.

Zu den Unterprogrammen: Der Parser sucht einen Text ab Adresse in A0, der mit je einem Blank beginnen und enden muß. Das letzte Blank wurde vorab mit

```
move.b #' ',-1(a0,d0.l)
```

an das Ende der Kommandozeile gezwungen, übrigens ein schönes Beispiel für die Mächtigkeit der 68000-Adressierungsarten. Der vom Parser isolierte Text wird in den Puffer ab A1 kopiert. Da es sich um Filenamen handelt, werden die Texte mit einem Null-Byte terminiert.

Die Routine »hex« konvertiert eine Binärzahl in einen String, der den Wert in hex darstellt. Wenn Sie den Schleifenzähler wieder von 3 in 7 ändern und das »rol« in »rol.l« können Sie damit auch Langworte wandeln, wie schon im Kapitel 5 erklärt.

13.4 Die Parameterübergabe

Bis hierher war die Sache zwar einfach, aber kaum realistisch. In der Regel wird man nämlich Daten an die Assembler-Routine übergeben müssen und sich das Ergebnis (die Ergebnisse) auch von dort holen wollen. Frage also: Wie übergibt man Parameter in beiden Richtungen?

Antwort 1: Man muß auch die Ergebnis-Variablen übergeben. Das Assembler-Programm muß diese dann ändern. Genauer: Man übergibt die Adressen der BASIC-Variablen, die Maschinenroutine schreibt ab dort das Ergebnis hinein.

Antwort 2: Die Parameterübergabe läuft über den Stack.

Am besten erklärt man das an einem Beispiel. Ich möchte ganz simpel, daß die Assembler-Routine in einem String das letzte Zeichen duch ein x ersetzt. Dazu muß ich zwei Parameter übergeben, nämlich die Adresse des Strings und seine Länge. In BASIC sähe das so aus wie in Bild 13.5.

```
DIM  a%(13),r%(3)
FOR i=1 TO 13
  READ a$:a%(i)=VAL("&h"+a$)
NEXT

a$="hallo"
Adresse&=SADD(a$)
Laenge&=LEN(a$)

ass&=VARPTR(a%(1))
CALL ass&(Adresse&,Laenge&)
PRINT a$

END

DATA 48E7,8080,206F,000C,202F,0010,5380,11BC
DATA 0078,0800,4CDF,0101,4E75
```

Bild 13.5: BASIC-String soll modifiziert werden

Wie üblich wird wieder die Routine aus Data-Zeilen geladen, ihre Adresse steht dann in ass&. Adresse und Länge des Strings werden bestimmt, und dann erfolgt der Aufruf mit

```
        CALL ass&(Adresse&,Laenge&)
```

Schauen wir uns nun an, was in den Data-Zeilen steckt; Bild 13.6 verrät das Geheimnis.

```
        opt     l+,p+
        movem.l a0/d0,-(sp)       ;8 mehr auf dem Stack
;                                  +Returnadresse macht 12

        move.l  12(sp),a0         ;Adresse
        move.l  16(sp),d0         ;Laenge
        subq.l  #1,d0
        move.b  #'x',0(a0,d0.l)

        movem.l (sp)+,a0/d0
        rts
```

Bild 13.6: Bedeutung der Data-Zeilen in Bild 13.5

Ich muß mir eigentlich nur merken, daß durch das »movem« zusätzlich noch 8 Byte auf den Stack gekommen sind (zwei Register) und auch noch die Return-Adresse mit 4 Byte existiert, macht in der Summe 12.

Nun kann ich einfach im 4er Abstand, beginnend mit 12, auf die Parameter der Reihe nach zugreifen. Doch Vorsicht, so einfach ist das nur, wenn alle Parameter vom Typ Long sind. Da der Stack von den hohen zu den tiefen Adressen wächst, ist 12(sp) tiefer als 16(sp). Das heißt, der Parameter bei 12(sp) ist der letzte auf dem Stack, dieser (Adresse&) war aber der erste im Aufruf. Anders ausgedrückt: Die Parameter werden in umgekehrter Reihenfolge abgelegt.

Schauen wir uns deshalb noch ein Beispiel an. Laut Bild 13.7 soll ein Array oder ein Teil davon ganz schnell mit dem gleichen Wert in allen (oder den ausgewählten) Elementen geladen werden.

```
DIM  a%(15),r%(100)
FOR i=1 TO 15
  READ a$:a%(i)=VAL("&h"+a$)
NEXT

ass&=0

Laenge&=5
Wert%=123
Adresse&=VARPTR(r%(3))

ass&=VARPTR(a%(1))
CALL ass&(Adresse&,Laenge&,Wert%)
FOR i=1 TO 10
  PRINT r%(i)
NEXT

END

DATA 48E7,C080,206F,0010,202F,0014,322F,001A
DATA 5380,30C1,51C8,FFFC,4CDF,0103,4E75
```

Bild 13.7: Ein Array wird initialisiert

Übergeben muß ich die Adresse des Elements, ab dem der Array geladen werden soll (hier r%(3)), die Länge (Anzahl der Elemente) und natürlich den Wert. Der Wert ist

aber als einziger vom Typ Integer, sprich belegt nur 2 Byte. Doch schauen wir uns an, was BASIC dafür auf den Stack packt. Der Aufruf lautet wieder:

```
CALL ass&(Adresse&,Laenge&,Wert%)
```

Schauen wir uns die Assembler-Routine dazu an (Bild 13.8).

```
        opt     l+,p+
        movem.l aØ/dØ-d1,-(sp)   ;12 mehr auf dem Stack
;                                 +Returnadresse macht 16

        move.l  16(sp),aØ        ;Adresse
        move.l  2Ø(sp),dØ        ;Anzahl
        move.w  26(sp),d1        ;Wert
        subq.l  #1,dØ            ;- 1 wegen dbra
loop    move.w  d1,(aØ)+
        dbra    dØ,loop

        movem.l (sp)+,aØ/dØ-d1
        rts
```

Bild 13.8: Die Assembler-Routine zu Bild 13.7

Daß ich nun bei 16 anfangen muß, ist klar. Schließlich sind drei Register und die Return-Adresse mit je 4 Byte auf dem Stack. Doch warum finde ich den Wert bei 26(sp) und nicht bei 24(sp)? Offensichtlich läßt sich BASIC durch das %Zeichen (was short Integer heißen soll) nicht beeindrucken, und packt auch dann 4 Byte auf den Stack. Tatsächlich können Sie in BASIC auch »Wert&« schreiben und dann den Wert bei 24(sp) als Langwort abholen. Das ist an sich klarer, macht aber Probleme bei negativen Zahlen, da dann ja das Vorzeichen sozusagen in Bit 31 steht, was natürlich schiefgeht, wenn man nur die Bits 0 bis 15 in ein Wort übernimmt.

Wenn bei Ihren Experimenten etwas schiefgeht, schauen Sie sich auch Ihr BASIC-Programm an. Ich habe beispielsweise beim Testen der Routine einen bildschönen Absturz erlebt, weil ich anstatt r% a% geschrieben habe, was dann leider zur Folge hatte, daß ich mit »Wert« die Assembler-Routine überschrieben habe, als sie mitten in der Ausführung war.

Auch sollten Sie darauf achten, keine neuen Variablen in BASIC anzuziehen, wenn Sie vorher mit VARPTR oder SADD eine Adresse bestimmt haben. Die neue Variable kann die älteren verschieben, so daß dann deren Adressen nicht mehr stimmen. Am

einfachsten weist man vorab allen Variablen einen Wert zu. So sinnlos ist also »ass& = 0« in Bild 13.7 gar nicht.

13.5 CLI-Befehle in BASIC aufrufen

Zum Schluß eine nützliche Routine, die den Shell-Befehl von MS-BASIC (IBM-PC) auch für Amiga-BASIC zur Verfügung stellt. Wie man den Execute-Befehl des DOS dafür nutzt, wissen Sie schon. Wie man einen String übergibt, ist Ihnen auch nicht neu. Kombiniert man alles, entsteht das Programm von Bild 13.9.

```
        opt     l+,p+   ;linkbar, pos. indep.

* shell.s  Routine zum Aufruf von CLI-Befehlen in Basic
* (c) 1987 Peter Wollschlaeger

_SysBase         equ    4                   ;Basis von Exec
_LVOOpenLibrary  equ    -552                ;Library oeffnen
_LVOCloseLibrary equ    -414                ;Library schliessen
_LVORead         equ    -42                 ;Lese File
_LVOOpen         equ    -30                 ;Open File
_LVOClose        equ    -36
_LVOExecute      equ    -222                ;Execute CLI-Cmd

RELO   macro
        lea     \1(pc),a0                   ;Pgm muss lage-
        move.l  a0,\2                       ;unabhaengig werden
        endm

        movem.l d0-d6/a0-a6,-(sp)           ;14*4=56 Bytes
;                                            + Return-Adr. = 60
;
_main   lea     dosname(pc),a1              ;Name der DOS-Lib
        moveq   #0,d0                       ;Version egal
        move.l  _SysBase,a6                 ;Basis Exec
        jsr     _LVOOpenLibrary(a6)         ;DOS-Lib oeffnen
        tst.l   d0                          ;Fehler?
        beq     fini                        ;wenn Fehler, Ende
        move.l  d0,a6                       ;Zeiger merken

        RELO    name,d1                     ;Name DOS-Lib
        move.l  #1005,d2                    ;Status = gibt es
        jsr     _LVOOpen(a6)                ;nun oeffnen
        move.l  d0,d5                       ;Handle merken
        tst.l   d0                          ;Fehler?
        beq     fini                        ;wenn ja, abbrechen
```

```
        move.l  60(sp),a0               ;Adresse Basic-String
        move.l  64(sp),d0               ;Laenge

        subq.l  #1,d0                   ;-1 wegen dbra
        lea     buffer(pc),a1           ;dahin
loop    move.b  (a0)+,(a1)+             ;kopieren
        dbra    d0,loop
        move.b  #0,(a1)                 ;terminiere mit 0

        RELO    buffer,d1               ;Adresse CLI-Befehl
        clr.l   d2                      ;kein Input
        move.l  d5,d3                   ;Window-Handle
        jsr     _LVOExecute(a6)         ;CLI aufrufen

        move.l  d5,d1                   ;von CON lesen
        RELO    buffer,d2               ;in diesen Puffer
        move.l  #1,d3                   ;1 Zeichen
        jsr     _LVORead(a6)            ;Lesen aufrufen

        move.l  d5,d1                   ;CON wieder schliessen
        jsr     _LVOClose(a6)

        move.l  a6,a1                   ;DOS-Lib-Basis
        move.l  _SysBase,a6             ;Basis Exec
        jsr     _LVOCloseLibrary(a6)    ;Funktion "Schliessen"
fini    movem.l (sp)+,d0-d6/a0-a6

        rts                             ;Return zum CLI

dosname  dc.b   'dos.library',0
         cnop   0,2
name     dc.b   'CON:10/20/600/160/Mit beliebiger Taste -> Basic',0
         cnop   0,2
buffer   dc.b   'ABCDE'                 ;damit ich das finde
         end
```

Bild 13.9: Damit wird der Shell-Befehl emuliert

Im Prinzip tue ich nichts weiter, als den String, den BASIC übergibt, in einen Puffer zu kopieren und mit einem Null-Byte abzuschließen. Damit kann ich dann _LVOExecute aufrufen. Die Ausgabe erfolgt in einem eigenen Fenster. Dann wird auf eine Taste gewartet (ein Zeichen gelesen), und es geht zurück zum BASIC. Dessen Listing zeigt Bild 13.10.

```
' Shell-Befehl in Amiga-Basic
' (c) 1987 Peter Wollschlaeger

DIM  assem%(140): InitShell

shell "dir jh0:"     'Demo

END

SUB InitShell STATIC
  SHARED assem%()
  FOR i=1 TO 96
    READ a$:assem%(i)=VAL("&h"+a$)
  NEXT
END SUB

SUB shell(a$) STATIC
  SHARED assem%()
  Adresse&=SADD(a$): Laenge&=LEN(a$)
  ass&=VARPTR(assem%(1))
  CALL ass&(Adresse&,Laenge&)
END SUB

DATA 48E7,FEFE,43FA,007E,7000,2C79,0000,0004
DATA 4EAE,FDD8,4A80,6700,0066,2C40,41FA,0072
DATA 2208,243C,0000,03ED,4EAE,FFE2,2A00,4A80
DATA 6700,004C,206F,003C,202F,0040,5380,43FA
DATA 0080,12D8,51C8,FFFC,12BC,0000,41FA,0072
DATA 2208,4282,2605,4EAE,FF22,2205,41FA,0062
DATA 2408,263C,0000,0001,4EAE,FFD6,2205,4EAE
'                                ---- ----
DATA FFDC,224E,2C79,0000,0004,4EAE,FE62,4CDF
DATA 7F7F,4E75,646F,732E,6C69,6272,6172,7900
DATA 434F,4E3A,3130,2F32,302F,3630,302F,3136
DATA 302F,4D69,7420,6265,6C69,6562,6967,6572
DATA 2054,6173,7465,202D,3E20,4261,7369,6300
```

Bild 13.10: Der BASIC-Teil zu Bild 13.9

Etwas tricky ist die Geschichte mit dem Puffer gelöst. Da der rund 80 Zeichen groß sein sollte, ein »ds.b 80« aber die Data-Zeilen um 40 Worte mit »0000« verlängert hätte, steht in BASIC gar nichts zum Thema Puffer. Im Assembler-Listing hatte ich den Pufferbeginn mit »dc.b 'ABCDE'« markiert. Damit konnte ich die Stelle in den ursprünglich von ABC erzeugten Data-Zeilen leicht finden. Ab diesem Wort (AB) habe ich dann alle Datas in BASIC gelöscht. Der Array wurde aber mit 140 größer dimen-

sioniert als der Code (96 Worte) dies erfordert. Der Rest dient als Puffer. So löst man also das Problem.

Bliebe noch eines: In dieser Lösung wartet die Assembler-Routine immer auf einen Tastendruck, bevor sie zum BASIC zurückkehrt. Das ist natürlich sinnvoll, wenn man sich das Ergebnis von zum Beispiel einem Dir-Befehl auch ansehen will, es stört jedoch, wenn man »shell« in einem laufenden Programm benutzt, um zum Beispiel Dateien zu kopieren. Mein Vorschlag wäre: Ersetzen Sie einfach das »jsr _LVORead(a6)« durch »NOP NOP«. NOP heißt No Operation (tue nichts) und hat den Code 4E71.

Im Listing sind die beiden Datenworte unterstrichen, die Sie mit NOPs überschreiben müssen. Am besten lösen Sie das in BASIC. Schreiben Sie zwei Sub-Programme, Shell und ShellWarte. Das eine schreibt zuerst (numerisch) $4EAE, $FFD6 in die richtigen Elemente des Arrays, das andere zweimal $4E71.

Kapitel 14

Exec und DOS im Detail

14.1 Prozesse und Tasks

Bisher hatten wir immer nur von Tasks gesprochen. Im Gegensatz dazu stehen beim Amiga Prozesse, was etwas verwirrend ist, denn üblicherweise ist Prozeß die deutsche Übersetzung von Task. Ich muß das deshalb wieder zurückübersetzen. In diesem Sinn ist ein Prozeß ein Ober-Task. Praktisch hat das zur Folge: Unter einem Prozeß können verschiedene Tasks (quasi) gleichzeitig laufen. Ein Task selbst kann nur einen anderen Task aufrufen, der dann anstatt seiner läuft.

Wenn Sie sich einmal die Datenstrukturen im Anhang ansehen, werden Sie feststellen, daß ein Task-Kontroll-Block nur eine Untermenge (ein Auszug) eines Prozeß-Kontroll-Blocks ist. Sie werden selten auf diese Strukturen zugreifen; wenn, dann sollten Sie jedoch wissen, auf welche. Im Kapitel 9 hatten wir im PLB (deutsch: Prozeß-Leit-Block) nachgesehen, wo sich unser Task befindet. Der direkte Zugriff im Sinn von Ändern von Parametern dieser Strukturen ist immer riskant (zu viele Randbedingungen sind zu beachten). Deshalb gibt es auch für alle wichtigen Anwendungen Funktionen, zum Beispiel für das Ändern der Priorität eines Tasks.

14.2 Exec, der Boß

Exec ist die Abkürzung von Executive, was im amerikanischen soviel wie Chef (leitender Angestellter) bedeutet. Exec ist natürlich ein Prozeß, der immer (solange der Amiga eingeschaltet ist) läuft.

Multitasking

Exec selbst ist das Multitasking-System des Amiga. So ein System hat die Aufgabe, die Ressourcen eines Systems auf verschiedene Tasks zu verteilen. Ressourcen sind die CPU (der 68000), der Hauptspeicher und die Peripherie-Geräte. Unteilbare Ressourcen wie die CPU werden beim Amiga nach einem Prioritäten-Verfahren vergeben. Zuerst wird der Task mit der höheren Priorität abgearbeitet, dann der nächst niedere. Haben mehrere Tasks die gleiche Priorität, werden sie nach einem Zeitscheibenverfahren (Time-Sharing) in Intervallen bearbeitet. Durch Hardware-Interrupts bekommt Exec die Sache immer wieder in den Griff, auch wenn ein Task seine Tätigkeit nicht beenden möchte.

Ein Task hat drei Zustände, nämlich laufend, nicht laufend und wartend. Mit Rücksicht auf andere Tasks sollte ein Task möglichst oft in den Zustand wartend (auf Eingabe) gesetzt werden. Wie das geschieht, wurde im Kapitel 12 (Events) gezeigt. Wenn man nicht _LVOWait aufruft, sondern dauernd mit _LVOGetMsg (meistens umsonst) nachfragt, ob eine Nachricht anliegt, verbraucht das nur unnötig Zeit, die anderen

Tasks dann fehlt. Ich betone das extra, denn ich habe leider schon viele Listings gesehen, die diesen unsauberen Programmierstil zeigen.

Speicherverwaltung

Die Ressource RAM wird jedem Task permanent zugeteilt, was seinen Speicher für Programm-Code und Daten betrifft. Gefährlich wird es für einen Task, wenn er dynamisch (während des Programmlaufs) Speicher anfordert (allokiert). Der Speicherbedarf kann unter Umständen nicht erfüllt werden, wenn andere Tasks schon alles verbraucht haben. Deshalb sollten Sie Ihren Speicherbedarf zum Programmanfang belegen (zu belegen versuchen) und nicht erst den User diverse Eingaben vornehmen lassen.

Sinngemäßes gilt für die Libraries. Sie müssen eine Library mit _LVOOpenLibrary öffnen. Exec schaut dann nach, ob diese Lib schon geladen ist (oder im ROM oder Kickstart-RAM steht). Wenn nicht, versucht Exec die Lib von der Disk zu laden. Fehlt dafür der Speicher, gibt Exec als Lib-Basis Null zurück. Folglich sollte Ihr Programm auch alle Libs zu Anfang öffnen und nicht erst bei Bedarf, wie man es manchmal sieht. Da könnte es dann zu spät sein. Auf jeden Fall sollten Sie aber eine Lib immer sofort schließen, wenn Sie sie nicht mehr brauchen. Sind Sie nämlich der einzige oder der letzte User dieser Lib, wird Exec den damit belegten Speicher wieder freigeben.

14.3 DOS, Workbench, Intuition, Libraries und Devices

Das Amiga-Betriebssystem ist sehr schön in verschiedene Module gegliedert, die hierarchisch einander zugeordnet sind. Bild 14.1 versucht, diesen Zusammenhang zu verdeutlichen.

Für Sie ist zuerst wichtig, daß Ihre Anwendung immer den Weg zur Hardware finden muß. So muß ein Workbench-Programm zuerst Intuition öffnen, um zum Beispiel ein Window erstellen zu können. Wollen Sie in diesem Window zeichnen, brauchen Sie zumindest Graphics, müssen also auch diese Lib öffnen.

AmigaDOS ist übrigens ein Prozeß, unter dem mehrere Tasks laufen. So läuft zum Beispiel immer der Disk-Validator, ein Task der ständig durch Probelesen nachsieht, ob in allen Laufwerken eine Diskette eingelegt ist.

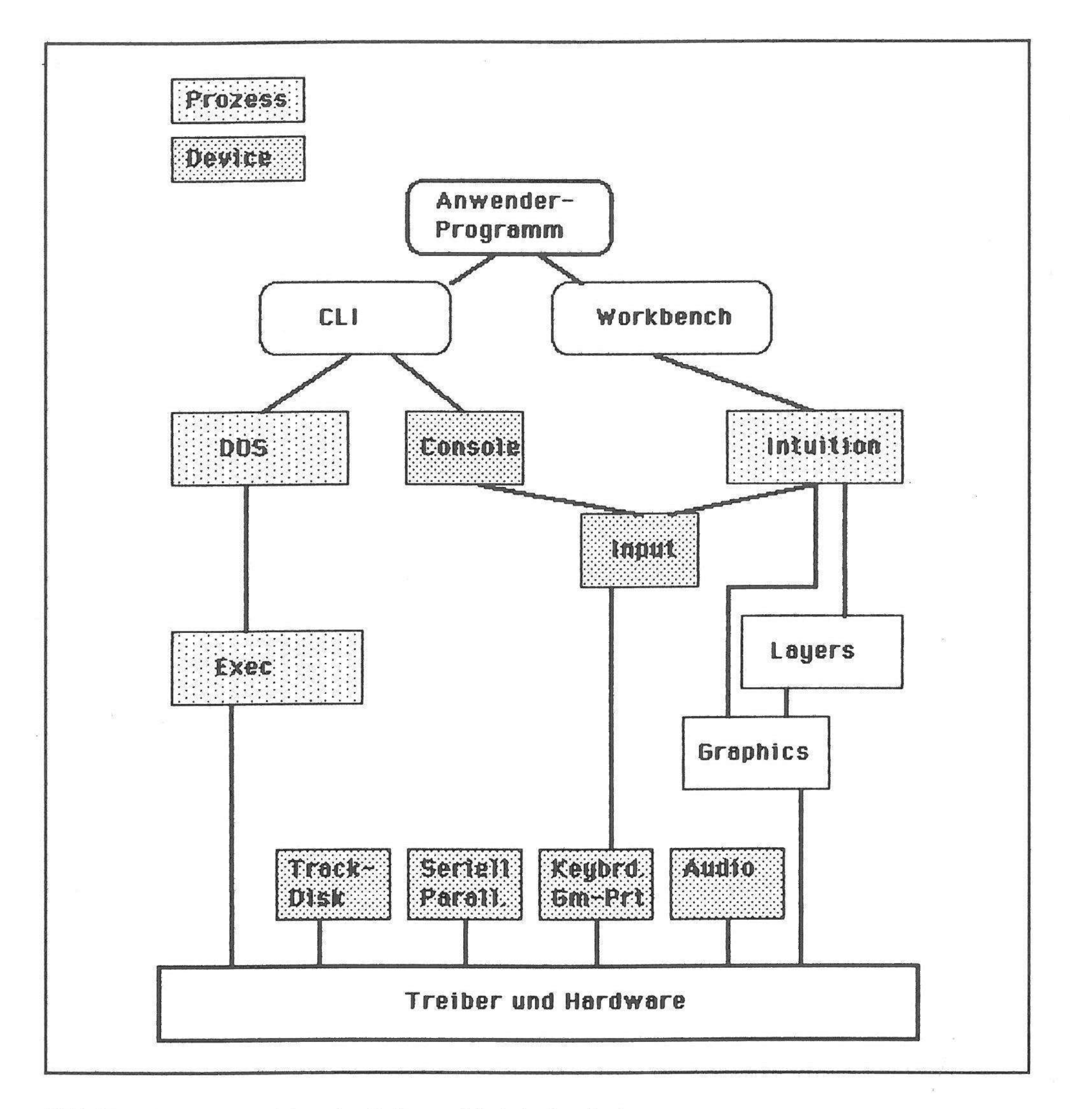


Bild 14.1: Zusammenwirken der Software-Module des Amiga

Workbench und CLI sind gleichberechtigte Bedienoberflächen, praktisch Programme, die die Eingaben des Anwenders interpretieren und im Rahmen ihrer Möglichkeiten ausführen. Mit welcher von beiden der Amiga startet, hängt lediglich vom Text in »startup-sequence« ab, womit übrigens der Amiga (auch) dem Atari ST überlegen ist, bei dem sich diese Start-Alternative nur durch direktes Ändern des Boot-Sektors der Startdiskette erreichen läßt, was solide Systemkenntnisse voraussetzt.

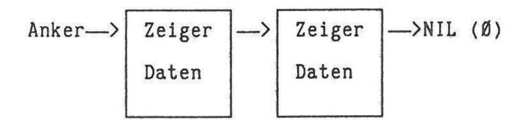
Intuition brauche ich Ihnen wohl nicht mehr vorzustellen, hier sollte nur die Einordnung gezeigt werden.

Devices

Die Schnittstelle zur Hardware bilden die Devices, schwach übersetzt, Geräte-Treiber. Denn auch diese Treiber sind Tasks, und damit herkömmlichen Treibern deutlich überlegen. Ein normaler Peripherie-Treiber stellt nur die Schnittstelle zur Verfügung, sprich übersetzt zum Beispiel DOS-Befehle in Bitmuster, die eine spezielle Hardware sehen will. Im DOS muß man dann immer noch zusehen, daß man Zeichen für Zeichen an zum Beispiel den Drucker los wird. Anders beim Amiga: Hier sagt man im Prinzip: Task (Device), das ist Deine Aufgabe, nun mach Du mal, ich mache inzwischen was anderes.

Strukturen und Listen

Wie Sie schon mitbekommen haben, läuft beim Amiga ohne Strukturen gar nichts (siehe auch Kapitel 11). Fast alle Strukturen werden in Form von Listen verwaltet, wie es das folgende Bild andeutet.



So ein Listenelement nennt man Knoten. Ein Knoten besteht immer aus (mindestens) einem Zeiger, der auf den nächsten Knoten zeigt, und Daten. Der letzte Knoten zeigt auf NIL (nichts), in Assembler wird dafür einfach eine Null eingetragen. Sie kennen solche Strukturen schon von zum Beispiel den Menüs her.

Exec verwaltet auf diese Art auch die Tasks. Jeder Task hat einen TCB (Task Control Block), dessen Daten in einer solchen Liste angelegt sind. Der Unterschied ist lediglich, daß die Exec-Listen doppelt verzeigert sind, sprich immer noch ein Zeiger auf den Vorgänger existiert. Damit lassen sich typische Operationen zur Task-Verwaltung wie das Sortieren der Tasks nach Prioritäten oder das Einfügen und Ausfügen von Knoten (Tasks) schneller realisieren. Der interessierte Leser sei auf den Include-File »nodes.i« (bei Metacomco und HiSoft) hingewiesen, wo man optimale Makros zur Listenverwaltung findet.

14.4 DOS und Exec in der Praxis

14.4.1 Directory

Zuerst möchte ich noch etwas aus dem DOS nachholen, was bisher nicht erwähnt wurde, nämlich der Umgang mit Locks und »FIBs«.

Der volle Zugriff auf einen File oder ein Directory ist nur über den FIB (File Info Block) möglich. Um auf diesen FIB zugreifen zu können, muß man sich vorher ein Lock (Schlüssel) besorgen, mit dem man den FIB aufschließen kann.

Schauen wir uns einmal den FIB an, hier als einen Auszug aus dem Anhang 4, wo Sie noch mehr Informationen finden können.

```
00      fib_DiskKey      ds.l    1
04      fib_DirEntryType ds.l    1          ;0=File, >0 = Dir.
08      fib_FileName     ds.b    108        ;trotzdem nur max. 30
74      fib_Protection   ds.l    1          ;siehe equ unten
78      fib_EntryType    ds.l    1
7C      fib_Size         ds.l    1          ;Filegroesse
80      fib_NumBlocks    ds.l    1
84      fib_DateStamp    ds.b    ds_SIZEOF ;letzte Aenderung
90      fib_Comment      ds.b    116
        fib_SIZEOF       equ     $104
```

Bild 14.2: Struktur des File Info Block

Sie sehen, hier stehen alle Informationen zu einem File, wie sein Name, seine Größe oder sein Typ. Die Struktur von »date_stamp« (Erstellungs-/Änderungsdatum) steht auch im Anhang 4.

Wie man mit diesen Informationen umgeht, zeigt man am besten mit einem kleinen Programm, das ein Directory auslistet. Hierzu dient Bild 14.3. Nun sagen Sie nicht, dafür könne man auch DIR im CLI tippen. Was machen Sie zum Beispiel in einem Programm, das prüfen soll, ob ein Directory oder File existiert?

```
      opt     l-                  ;nicht linken!

* DIR_2 Directory mittels DOS-Funktionen anzeigen

_LVOLock        equ     -84
_LVOExamine     equ     -102
_LVOExNext      equ     -108
_LVOIoErr       equ     -132
ERROR_NO_MORE_ENTRIES   equ     232

        include OpenDos.i               ;siehe Kapitel 4

        jsr     _LVOOutput(a6)          ;Output-Handle holen
        move.l  d0,d4                   ;und merken

        move.l  #pfad,d1                ;Adresse Pfadname
        move.l  #-2,d2                  ;Zugriff Lesen
        jsr     _LVOLock(a6)            ;Lock zum Dir holen
        tst.l   d0                      ;gibt es Dir?
        beq     fertig                  ;wenn nicht
        move.l  d0,d5                   ;sonst Lock merken

        move.l  d5,d1                   ;Lock
        move.l  #fib,d2                 ;Adresse FIB
        jsr     _LVOExamine(a6)         ;1. Namen (hier Disk) holen
        tst.l   d0                      ;Gefunden?
        beq     fertig                  ;wenn nicht
        bsr     print                   ;sonst ausgeben

loop    move.l  d5,d1                   ;Lock
        move.l  #fib,d2                 ;FIB
        jsr     _LVOExNext(a6)          ;Naechsten File suchen
        tst.l   d0                      ;gefunden?
        beq     fertig                  ;wenn nicht
        bsr     print                   ;sonst ausgeben
        bra     loop                    ;bis nichts mehr

fertig  jsr     _LVOIoErr(a6)           ;Error-Code holen
        cmpi.l  #NO_MORE_ENTRIES,d0     ;dieser Fehler?
        beq     f1                      ;wenn so

        move.l  d4,d1                   ;Output-Handle
        move.l  #err,d2                 ;Text
        move.l  #err_len,d3             ;d3
        jsr     _LVOOutput(a6)          ;Ausgabe

f1      move.l  a6,a1                   ;DOS-Lib schliessen
        move.l  _SysBase,a6
        jsr     _LVOCloseLibrary(a6)
fini    rts
```

```
print   move.l  #fib+8,a0            ;Adresse Name
        clr.l   d3                   ;Laengenzaehler
p1      addq.l  #1,d3                ;inkrementieren
        tst.b   (a0)+                ;Null-Byte?
        bne     p1                   ;wenn nicht
        move.b  #10,(a0)             ;0 durch LF ersetzen
        move.l  d4,d1                ;Output-Handle
        move.l  #fib+8,d2            ;Adresse Name
        jsr     _LVOOutput(a6)       ;Name ausgeben
        rts

* Datenbereich

dosname dc.b    'dos.library',0
        cnop    0,2
pfad    dc.b    'df0:',0
        cnop    0,2
fib     ds.b    $104                 ;siehe Anhang A4.2
                                     ;FileInfoBlock
err     dc.b    10,'Es ging was schief',10
err_len equ     *-err
```

Bild 14.3: Zugriff auf Directory

Den Lock erhält man sehr einfach auf diese Art:

```
        move.l  #pfad,d1             ;Adresse Pfadname
        move.l  #-2,d2               ;Zugriff Lesen
        jsr     _LVOLock(a6)         ;Lock zum Dir holen
```

Wie üblich, steht nun der Lock in d0, wir sichern ihn gleich in d5. Nun kann man Funktionen aufrufen, die als Parameter den Lock und die Adresse des FIB (bisher nur ein leerer Speicherbereich) erwarten. Ein Beispiel dafür wäre (Lock in d5):

```
        move.l  d5,d1                ;Lock
        move.l  #fib,d2              ;Adresse FIB
        jsr     _LVOExamine(a6)      ;1. Namen (hier Disk) holen
```

_LVOExamine prüft, ob der Pfad (hier der Disk-Name) existiert und schreibt ihn im Ja-Fall in den FIB. Hier steht nun der Name ab FIB+8 (siehe Bild 14.2). Wir wissen nicht, wie lang der Name ist, sondern nur, daß er mit 0-Byte endet. Die Routine »print« zählt daher zuerst die Zeichen bis zum 0-Byte und ruft dann das schon bekannte _LVOWrite auf.

Nun gibt es wie in jedem DOS eine Funktion, die weitere Einträge sucht, hier heißt sie _LVOExNext (Examine Next). Gibt es nichts mehr, »returnt« sie Null, sonst können

wir den Namen ausdrucken. Versuchen Sie doch einmal, auch die File-Größe auszugeben. Mit

```
        move.l  fib+$7C,d2
```

bringen Sie den Wert in d2, wo ihn »Bindec« (kennen Sie noch?) erwartet. Vergessen Sie aber nicht, das LF in »print« herauszunehmen und nach der Zahlenausgabe das LF auszugeben. Ob es sich bei dem Namen um ein Directory handelt, läßt sich mittels »EntryTyp« feststellen.

14.4.2 CLI-Befehle aufrufen

Wenn Sie das Directory nur anzeigen wollen, geht das vom Programm aus allerdings noch einfacher. Man kann nämlich jeden CLI-Befehl, also auch DIR, von einem Programm her aufrufen. Bleiben wir gleich bei DIR, Bild 14.4 zeigt die Lösung.

```
        opt     l-              ;nicht linken!

* DIR_1 Directory mittels Execute anzeigen

_LVOExecute     equ     -222
        include OpenDos.i               ;siehe Kapitel 4

        move.l  #string,d1              ;Adresse CLI-Befehl
        clr.l   d2                      ;kein Input
        clr.l   d3                      ;Output im CLI-Window
        jsr     _LVOExecute(a6)         ;CLI aufrufen

        move.l  a6,a1                   ;DOS-Lib schliessen
        move.l  _SysBase,a6
        jsr     _LVOCloseLibrary(a6)

fini    rts

* Datenbereich

dosname dc.b    'dos.library',Ø
        cnop    Ø,2
string  dc.b    'dir',Ø
```

Bild 14.4: Aufruf von CLI-Befehlen (hier DIR) aus einem Programm

Die Funktion _LVOExecute verlangt als Parameter zuerst die Adresse (in d1), ab der der Befehl im Klartext steht. In d2 und d3 müssen Nullen stehen oder File-Handles, wie man sie von _LVOOpen erhält. Steht in d2 die Handle einer Eingabedatei, wird diese Datei gelesen und ihr Inhalt als Befehlssequenz interpretiert.

Nun kommt der Knüller: Zuerst wird natürlich der mit d1 bezeichnete Befehl ausgeführt. Da muß aber keiner stehen, Sie können auch d1 nullen. Das heißt, daß nun nur die Kommandosequenz in dem mit d2 bezeichneten File ausgeführt wird. Wissen Sie jetzt, was der EXECUTE-Befehl des CLI macht?

Steht in d3 eine Null, erfolgt die Ausgabe im aktuellen CLI-Fenster, ansonsten im File, dessen Handle hier eingetragen ist.

14.4.3 Exec

Zwischen DOS und Exec besteht ein enger Zusammenhang. Zuerst wird natürlich auch DOS von Exec kontrolliert, zum anderen sind viele DOS-Befehle recht faul, Sie reichen die Arbeit nur an Exec weiter.

Das gilt für das gesamte IO (Input, Output), egal ob Disketten, der Drucker oder die Sprachausgabe gemeint sind. Prinzipiell läuft das IO so ab, daß ein IO-Request-Block mit Parametern geladen wird. Dann wird ein Device-Task aufgefordert (requested), den Job zu erledigen. Exec schiebt die Arbeit also auch nur weiter. Der Ablauf ist also immer folgender:

1. Request-Block initialisieren
2. Device öffnen
3. DoIO oder SendIO aufrufen
4. Device schließen

Bei DoIO wird der Request an den Device-Task geschickt und dann gewartet, bis das Ergebnis vorliegt (Funktionswert O.K., Ergebnis im Puffer). Bei SendIO passiert zuerst das gleiche, nur wird dann nicht gewartet. Das Programm läuft weiter und der Task dazu parallel. Das Thema möchte ich in diesem Buch nicht mehr vertiefen, weil es nur für Spezialfälle von Interesse ist. Für den Normalfall stehen komfortable Funktionen und fertige Routinen zur Verfügung.

Anhang

Anhang A1: Befehlsliste des 68000

Die Befehle sind in einem kompakten Format wie im folgenden Beispiel dargestellt:

ADD ea,Dn / ADD Dn,ea — Addiere
B W L — S+D → D

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
S:	*	*	WL	*	*	*	*	*	*	*	*	*
D:			*		*	*	*	*	*	*		

In der ersten Zeile steht immer der Befehl in den erlaubten Syntax-Formen gefolgt von einer kurzen Erklärung.

Dn ist ein beliebiges Datenregister
An ist ein beliebiges Adreßregister
Rn ist ein beliebiges Adreß- oder Datenregister
S ist der Source-(Quell-) Operand
D ist der Destination-(Ziel-) Operand

#K ist eine Konstante
d ist die Adreßdistanz

In der zweiten Zeile stehen die erlaubten Operandengrößen (B, W, L). Darunter stehen die möglichen Adressierungsarten. Ein »*« heißt, daß alle der vorgenannten Operandengrößen auch bei dieser Adressierung erlaubt sind. Ein oder zwei Buchstaben beschränken die Adressierungsart auf diese Operandengrößen.

cc in zum Beispiel DBcc heißt Condition Code. Seine Bedeutung ist auf der letzten Seite dieses Anhangs aufgeführt.

ABCD Dn,Dn / ABCD -(An),-(An) — Addiere BCD
B — S+D+X → D

ADD ea,Dn / ADD Dn,ea — Addiere

B W L — S+D → D

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
S:	*	*	WL	*	*	*	*	*	*	*	*	*
D:			*	*	*	*	*	*	*			

ADDA ea,An — Addiere Adresse

W L — S+D → D

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
S:	*	WL	*	*	*	*	*	*	*	*	*	*

Wortoperand wird wie bei EXT.L erweitert

ADDI #K,ea — Addiere Konstante

B W L — #K+D → D

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:	*		*	*	*	*	*	*	*			

ADDQ #K,ea — Addiere Konstante Quick (#K < =8)

B W L — #K+D → D

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:	*	WL	*	*	*	*	*	*	*			

ADDX Dn,Dn / ADDX -(An),-(An) — Addiere mit X-Flag

S+D+X → D

AND ea,Dn / AND Dn,ea — Logisch UND

B W L — S AND D → D

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
S:	*			*	*	*	*	*	*	*	*	*
D:			*	*	*	*	*	*	*			

ANDI #K,ea — Logisch UND mit Konstante

B W L — #K AND D → D

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:	*		*	*	*	*	*	*	*			

ANDI #K,CCR — Unde zu CCR

B — #K AND CCR → CCR

ANDI #K,SR — Unde zu SR — ! Privilegiert !

W — #K AND SR → SR

ASL Dn,Dn / ASL #K,Dn / ASL ea — Arithmetisch links schieben

B W L — D n Bits geschoben → D

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:			*	*	*	*	*	*	*			

0 wird nachgeschoben, herausgeschobenes Bit geht in das C- und X-Flag

---

ASR Dn,Dn / ASR #K,Dn / ASR ea — Arithmetisch rechts schieben

B W L — D n Bits geschoben → D

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:			*	*	*	*	*	*	*			

MS-Bit schiebt, bleibt aber erhalten. Herausgeschobenes Bit geht in das C- und X-Flag

---

Bcc Label — Verzweige, wenn cc (PC-relativ)

.B W — siehe cc-Tabelle

.S — PC+d → PC

BCHG Dn,ea / BCHG #K,ea — Bit n testen und ändern
B L — Bit-Test → Z-Flag
Bit ändern

Wenn Source Dn: n=0..31, sonst 0..7. Wenn Destination im RAM, wird immer 1 Byte gelesen und n=n mod 8.

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:	L		B	B	B	B	B	B	B			

BCLR Dn,ea / BCLR #K,ea — Bit n testen und löschen
B L — Bit-Test → Z-Flag
Bit = 0

Wenn Source Dn: n=0..31, sonst 0..7. Wenn Destination im RAM, wird immer 1 Byte gelesen und n=n mod 8.

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:	L		B	B	B	B	B	B	B			

BRA Label — Verzweige zu Label (PC-relativ)
.B W — PC+d → PC
.S

BSET Dn,ea / BSET #K,ea — Bit n testen und setzen
B L — Bit-Test → Z-Flag
Bit = 1

Wenn Source Dn: n=0..31, sonst 0..7. Wenn Destination im RAM, wird immer 1 Byte gelesen und n=n mod 8.

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:	L		B	B	B	B	B	B	B			

BSR Label — Call Sub bei Label (PC-relativ)
.B W — PC → -(SP); PC+d → PC
.S

BTST Dn,ea / BSET #K,ea — Bit n testen
B L — Bit-Test → Z-Flag

Wenn Source Dn: n=0..31, sonst 0..7. Wenn Destination im RAM, wird immer 1 Byte gelesen und n=n mod 8.

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
S:	L		B	B	B	B	B	B	B	B	B	B
D:	L		B	B	B	B	B	B	B	B	B	

CHK ea,Dn — Register gegen Limits checken
W — if Dn <0 or Dn >(ea) then trap

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
S:	W		W	W	W	W	W	W	W	W	W	W

CLR ea — Lösche Operand
B W L — 0 → D

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:	*		*	*	*	*	*	*	*			

CMP ea,Dn — Vergleiche Operanden
B W L — Flags wie nach D minus S

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
S:	*	W	*	*	*	*	*	*	*	*	*	*

CMPA ea,An — Vergleiche Adressen
W L — Flags wie nach D minus S

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
S:	*	*	*	*	*	*	*	*	*	*	*	*

Wort-Operand wird vorher auf Long erweitert

CMPI #K,ea — Vergleiche gegen Konstante
B W L — Flags wie nach D minus S

	Dn	An	(An)	(An)+	-(An)	(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:	*		*	*	*	*	*	*	*			

CMPM (An)+,(An)+ — Vergleiche Speicherstellen
B W L — Flags wie nach D minus S

DBcc Dn,Label — Teste cc. Dekrementiere Dn. Branch
if cc = false then Dn=Dn-1
if Dn < > -1 then BRA Label
else »hier weiter«

DIVS ea,Dn — Dividiere Worte Signed
W — D/S → D

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
S:	*		*	*	*	*	*	*	*	*	*	*

Quotient im niederwertigen Wort, Rest im höherwertigen

DIVU ea,Dn — Dividiere Worte Unsigned
W — D/S → D

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
S:	*		*	*	*	*	*	*	*	*	*	*

Quotient im niederwertigen Wort, Rest im höherwertigen

EOR Dn,ea — Logisch XOR
B W L — S xor D → D

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:	*		*	*	*	*	*	*	*			

EORI #K,ea — Logisch XOR mit Konstante
B W L — S xor D → D

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:	*		*	*	*	*	*	*	*			

EORI #K,CCR — XOR Konstante mit CCR
B — S xor CCR → CCR

EORI #K,SR — XOR Konstante mit SR !Privileg. !
W — S xor CCR → CCR

EXG Rn,Rn — Tausche Register
L — Rn ↔ Rn

EXT Dn — Dn vorzeichenrichtig erweitern
W L

ILLEGAL — löst Illegal-Exception aus

JMP ea — absoluter Sprung (lang)
D → PC

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:			*			*	*	*	*	*	*	

JSR ea — absoluter UP-Aufruf
PC → -(SP); D → PC

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:			*			*	*	*	*	*	*	

LEA ea,An
L

Lade effektive Adresse
D → An

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:			*			*	*	*	*	*	*	

LINK An,#d

Lokalen Stack einrichten
An → -(SP); SP → An; SP+d → SP

LINK und UNLK werden gebraucht, um eine »linked list« von lokalen Variablen für verschachtelte UP-Aufrufe anzulegen

---

LSL Dn,Dn / LSL #K,Dn / LSL ea
B W L

Logisch links schieben
D n Bits geschoben → D

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:			*	*	*	*	*	*	*			

0 wird nachgeschoben, herausgeschobenes Bit geht in das C- und X-Flag

---

LSR Dn,Dn / LSR #K,Dn / LSR ea
B W L

Logisch rechts schieben
D n Bits geschoben → D

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:			*	*	*	*	*	*	*			

0 wird nachgeschoben, herausgeschobenes Bit geht in das C- und X-Flag

---

MOVE ea,ea
B W L

Kopiere Daten
D → S

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
S:	*	WL	*	*	*	*	*	*	*	*	*	*
D:	*		*	*	*	*	*	*	*			

MOVE CCR,ea — CCR holen
W — CCR → ea

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:	*		*	*	*	*	*	*	*			

MOVE ea,CCR — CCR laden
W — ea → CCR

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
S:	*		*	*	*	*	*	*	*	*	*	*

MOVE ea,SR — SR laden — ! Privilegiert !
W — ea → SR

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
S:	*		*	*	*	*	*	*	*	*	*	*

MOVE SR,ea — SR holen — ! Privilegiert !
W — SR → ea

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:	*		*	*	*	*	*	*	*			

MOVE USP,An — USP holen — ! Privilegiert !
L — USP → An

MOVEA ea,An — Kopiere Adresse
L — ea → An

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
S:	*	*	*	*	*	*	*	*	*	*	*	*

MOVEM R_Liste,ea / MOVEM ea,R_Liste — Registerliste kopieren
W L

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
1:			*		*		*	*	*			
2:			*	*		*	*	*	*	*	*	

1= Register → Speicher z.B.: movem d1-d3/a1-a4,-(a7)
2= Speicher → Register z.B.: movem (a7)+,d1-d3/a1-a4

MOVEP Dn,d(An) / MOVEP d(An),Dn — Daten von/zu Peripherie
W L
Daten werden byteweise übertragen

MOVEQ #K,Dn — Übertrage »Quick«
L — #K(8 Bit) → Dn

MULS ea,Dn — Multipliziere Signed
W — S*D → D

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
S:	*		*	*	*	*	*	*	*	*	*	*

MULU ea,Dn — Multipliziere Unsigned
W — S*D → D

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
S:	*		*	*	*	*	*	*	*	*	*	*

NBCD ea — Negiere BCD-Zahl
B — 0-D-X → D

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:	*		*	*	*	*	*	*	*			

NEG ea — Negiere Operand

B W L — $0 - D \rightarrow D$

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:	*		*	*	*	*	*	*	*			

NEGX ea — Negiere Operand mit X-Flag

B W L — $0 - D - X \rightarrow D$

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:	*		*	*	*	*	*	*	*			

NOP — No Operation

tue nichts (dauert vier Clock-Zyklen)

NOT ea — Logisch Nicht

B W L — $-D \rightarrow D$ (Einer-Komplement)

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:	*		*	*	*	*	*	*	*			

OR ea,Dn/ OR Dn,ea — Logisch ODER

B W L — S or D → D

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
S:	*		*	*	*	*	*	*	*	*	*	*
D:			*	*	*	*	*	*	*			

ORI #K,ea — Logisch ODER mit Konstante

B W L — #K or D → D

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:	*		*	*	*	*	*	*	*			

ORI #K,CCR — Odere zu CCR
B — #K or CCR → CCR

ORI #K,SR — Odere zu SR — ! Privilegiert !
W — #K or SR → SR

PEA ea — Push effektive Adresse
L — D → -(SP)

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:			*			*	*	*	*	*	*	

RESET — Rücksetzen — ! Privilegiert !
Reset-Leitung für 124 Clock-Zyklen auf 0

ROL Dn,Dn / ROL #K,Dn / ROL ea — Rotiere links
B W L — D n Bits rotiert → D

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:			*	*	*	*	*	*	*			

MS-Bit geht ins LS-Bit und ins Carry-Flag und schiebt links

ROR Dn,Dn / ROR #K,Dn / ROR ea — Rotiere rechts
B W L — D n Bits rotiert → D

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:			*	*	*	*	*	*	*			

LS-Bit geht ins MS-Bit und ins Carry-Flag und schiebt rechts

ROXL Dn,Dn / ROXL #K,Dn / ROXL ea — Rotiere links mit X-Flag
B W L — D n Bits rotiert → D

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:			*	*	*	*	*	*	*			

X geht ins LS-Bit und schiebt links. MS-Bit geht in X und Carry

---

ROXR Dn,Dn / RORL #K,Dn / RORL ea — Rotiere rechts mit X-Flag
B W L — D n Bits rotiert → D

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:			*	*	*	*	*	*	*			

X geht ins MS-Bit und schiebt rechts. LS-Bit geht in X und Carry

---

RTE — Return von Exception ! Privilegiert !
(Sp) + → SR; (SP) + → PC

RTR — Return mit Flags
(Sp) + → CCR; (SP) + → PC

RTS — Return
(SP) + → PC

SBCD Dn,Dn / ABCD -(An),-(An) — Subtrahiere BCD
B — D-S-X → D

Scc — Setze Byte, wenn cc true
if cc=true $FF → Byte
else $00 → Byte

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:	*		*	*	*	*	*	*	*			

STOP #K — Lade SR und Halt ! Privilegiert !
#K → SR; Halt bis Interrupt

SUB ea,Dn / ADD Dn,ea — Subtrahiere
B W L — D-S → D

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
S:	*	*	WL	*	*	*	*	*	*	*	*	*
D:			*	*	*	*	*	*	*			

SUBA ea,An — Subtrahiere Adresse
W L — D-S → D

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
S:	*	WL	*	*	*	*	*	*	*	*	*	*

Wortoperand wird wie bei EXT.L erweitert

SUBI #K,ea — Subtrahiere Konstante
B W L — D-#K → D

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:	*		*	*	*	*	*	*	*			

SUBQ #K,ea — Subtrahiere Konstante Quick (#K < =8)
B W L — D-#K → D

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:	*	WL	*	*	*	*	*	*	*			

SUBX Dn,Dn / ADDX -(An),-(An) — Subtrahiere mit X-Flag
D-S-X → D

SWAP Dn — Tausche Worte in Dn
W — Bit 31..16 ↔ Bit 15..0

TAS ea — Teste und setze Bit 7 im Byte
B — Bit 7 → N/Z-Flag
1 → Bit 7

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:	*		*	*	*	*	*	*	*			

TRAP #n — Trap-Exception
PC → -(SSP)
SR → -(SSP)
Vektor n → PC

TRAPV — Trap, wenn Overflow

TST ea — Teste Operand gegen Null
B W L — Ergebnis in N/Z-Flag

	Dn	An	(An)	(An)+	-(An)	d(An)	d(An,Rn)	$.W	$.L	d(PC)	d(PC,Rn)	#
D:	*		*	*	*	*	*	*	*			

UNLK An — Unlink
An → SP; (SP)+ → An

Die Bedeutung der Condition Codes

	Kürzel	Bedeutung	Deutsch
	CC	Carry Clear	Carry = 0
	CS	Carry Set	Carry = 1
	EQ	Equal	Z =1
	GE	Greater or Equal	>=
***	GT	Greater Than	>
	HI	Higher	>
***	LE	Less or Equal	<
	LS	Less or Same	<=
***	LT	Less Than	<
	MI	Minus	-
	NE	Not Equal	< >
	PL	Plus	+
***	VC	oVerflow Clear	V = 0
***	VS	oVerflow Set	V =1

*** Für vorzeichenbehaftete Zahlen

Anhang A2: Library Vector Offsets

In diesem Anhang finden Sie die _LVOs und zwar

A2.1 Exec-Library
A2.2 DOS-Library
A2.3 Intuition-Library
A2.4 Graphics-Library
A2.5 Icon-Library
A2.6 Die drei Mathematik-Libraries
A2.7 Sonstige (Diskfonts und Translator)

jeweils alphabetisch sortiert.

Hinweis: Die Namen der LVOs entsprechen denen der Amiga-Dokumentation. Daran halten sich die Assembler von Metacomco und HiSoft. Für den SEKA-Assembler müssen Sie den Unterstrich vor den LVOs und der Basis-Adresse weglassen. Die Basis-Adresse für Exec ist eine Konstante, das heißt _SysBase equ 4.

Für alle anderen Libs ist die Basis eine Variable. Für HiSoft und SEKA ist dafür mit dc.l Platz im Programm vorzusehen. Nach dem Öffnen ist d0 in diese Variable zu kopieren, wenn man die Standard-Makros anwenden will. Bei Metacomco reicht XDEF (und Linken mit amiga.lib).

A 2.1 Exec-Library

Name zum Öffnen: `exec.library` (wird nie benötigt)
Basis-Adresse: `_SysBase`

```
_LVOAbortIO          equ     -480
_LVOAddDevice        equ     -432
_LVOAddHead          equ     -240
_LVOAddIntServer     equ     -168
_LVOAddLibrary       equ     -396
_LVOAddPort          equ     -354
_LVOAddResource      equ     -486
_LVOAddTail          equ     -246
_LVOAddTask          equ     -282
_LVOAlert            equ     -108
_LVOAllocAbs         equ     -204
_LVOAllocate         equ     -186
_LVOAllocEntry       equ     -222
_LVOAllocMem         equ     -198
```

```
_LVOAllocTrap         equ     -342
_LVOAvailMem          equ     -216
_LVOCause             equ     -180
_LVOCheckIO           equ     -468
_LVOCloseDevice       equ     -450
_LVOCloseLibrary      equ     -414
_LVODeallocate        equ     -192
_LVODebug             equ     -114
_LVODisable           equ     -120
_LVODispatch          equ     -60
_LVODoIO              equ     -456
_LVOEnable            equ     -126
_LVOEnqueue           equ     -270
_LVOException         equ     -66
_LVOExitIntr          equ     -36
_LVOFindName          equ     -276
_LVOFindPort          equ     -390
_LVOFindResident      equ     -96
_LVOFindTask          equ     -294
_LVOForbid            equ     -132
_LVOFreeEntry         equ     -228
_LVOFreeMem           equ     -210
_LVOFreeSignal        equ     -336
_LVOFreeTrap          equ     -348
_LVOGetCC             equ     -528
_LVOGetMsg            equ     -372
_LVOInitCode          equ     -72
_LVOInitResident      equ     -102
_LVOInitStruct        equ     -78
_LVOInsert            equ     -234
_LVOMakeFunctions     equ     -90
_LVOMakeLibrary       equ     -84
_LVOOldOpenLibrary    equ     -408
_LVOOpenDevice        equ     -444
_LVOOpenLibrary       equ     -552
_LVOOpenResource      equ     -498
_LVOPermit            equ     -138
_LVOProcure           equ     -540
_LVOPutMsg            equ     -366
_LVORawDoFmt          equ     -522
_LVORawIOInit         equ     -504
_LVORawMayGetChar     equ     -510
_LVORawPutChar        equ     -516
_LVORemDevice         equ     -438
_LVORemHead           equ     -258
_LVORemIntServer      equ     -174
_LVORemLibrary        equ     -402
_LVORemove            equ     -252
_LVORemPort           equ     -360
_LVORemResource       equ     -492
_LVORemTail           equ     -264
_LVORemTask           equ     -288
```

```
_LVOReplyMsg            equ     -378
_LVOReschedule          equ     -48
_LVOSchedule            equ     -42
_LVOSendIO              equ     -462
_LVOSetExcept           equ     -312
_LVOSetFunction         equ     -420
_LVOSetIntVector        equ     -162
_LVOSetSignal           equ     -306
_LVOSetSR               equ     -144
_LVOSetTaskPri          equ     -300
_LVOSignal              equ     -324
_LVOAllocSignal         equ     -330
_LVOSumLibrary          equ     -426
_LVOSuperState          equ     -150
_LVOSupervisor          equ     -30
_LVOSwitch              equ     -54
_LVOTypeOfMem           equ     -534
_LVOUserState           equ     -156
_LVOVacate              equ     -546
_LVOWait                equ     -318
_LVOWaitIO              equ     -474
_LVOWaitPort            equ     -384
```

A 2.2 DOS-Library

Name zum Öffnen: `dos.library`
Basis-Adresse: `_DOSBase`

```
_LVOClose               equ     -36
_LVOCreateDir           equ     -120
_LVOCreateProc          equ     -138
_LVOCurrentDir          equ     -126
_LVODateStamp           equ     -192
_LVODelay               equ     -198
_LVODeleteFile          equ     -72
_LVODeviceProc          equ     -174
_LVODupLock             equ     -96
_LVOExamine             equ     -102
_LVOExecute             equ     -222
_LVOExit                equ     -144
_LVOExNext              equ     -108
_LVOGetPacket           equ     -162
_LVOInfo                equ     -114
_LVOInput               equ     -54
_LVOIoErr               equ     -132
_LVOIsInteractive       equ     -216
_LVOLoadSeg             equ     -150
_LVOLock                equ     -84
_LVOOpen                equ     -30
_LVOOutput              equ     -60
```

```
_LVOParentDir          equ     -210
_LVOQueuePacket        equ     -168
_LVORead               equ     -42
_LVORename             equ     -78
_LVOSeek               equ     -66
_LVOSetComment         equ     -180
_LVOSetProtection      equ     -186
_LVOUnLoadSeg          equ     -156
_LVOUnLock             equ     -90
_LVOWaitForChar        equ     -204
_LVOWrite              equ     -48
```

A 2.3 Intuition-Library

Name zum Öffnen: `intuition.library`
Basis-Adresse: `_IntuitionBase`

```
_LVOAddGadget          equ     -42
_LVOAllocRemember      equ     -396
_LVOAlohaWorkbench     equ     -402
_LVOAutoRequest        equ     -348
_LVOBeginRefresh       equ     -354
_LVOBuildSysRequest    equ     -360
_LVOClearDMRequest     equ     -48
_LVOClearMenuStrip     equ     -54
_LVOClearPointer       equ     -60
_LVOCloseScreen        equ     -66
_LVOCloseWindow        equ     -72
_LVOCloseWorkBench     equ     -78
_LVOCurrentTime        equ     -84
_LVODisplayAlert       equ     -90
_LVODisplayBeep        equ     -96
_LVODoubleClick        equ     -102
_LVODrawBorder         equ     -108
_LVODrawImage          equ     -114
_LVOEndRefresh         equ     -366
_LVOEndRequest         equ     -120
_LVOFreeRemember       equ     -408
_LVOFreeSysRequest     equ     -372
_LVOGetDefPrefs        equ     -126
_LVOGetPrefs           equ     -132
_LVOInitRequester      equ     -138
_LVOIntuition          equ     -36
_LVOItemAddress        equ     -144
_LVOLockIBase          equ     -414
_LVOMakeScreen         equ     -378
_LVOModifyIDCMP        equ     -150
_LVOModifyProp         equ     -156
_LVOMoveScreen         equ     -162
_LVOMoveWindow         equ     -168
```

```
_LVOOffGadget        equ    -174
_LVOOffMenu          equ    -180
_LVOOnGadget         equ    -186
_LVOOnMenu           equ    -192
_LVOOpenIntuition    equ    -30
_LVOOpenScreen       equ    -198
_LVOOpenWindow       equ    -204
_LVOOpenWorkBench    equ    -210
_LVOPrintIText       equ    -216
_LVORefreshGadgets   equ    -222
_LVORemakeDisplay    equ    -384
_LVORemoveGadget     equ    -228
_LVOReportMouse      equ    -234
_LVORequest          equ    -240
_LVORethinkDisplay   equ    -390
_LVOScreenToBack     equ    -246
_LVOScreenToFront    equ    -252
_LVOSetDMRequest     equ    -258
_LVOSetMenuStrip     equ    -264
_LVOSetPointer       equ    -270
_LVOSetPrefs         equ    -324
_LVOIntuiTextLength  equ    -330
_LVOSetWindowTitles  equ    -276
_LVOShowTitle        equ    -282
_LVOSizeWindow       equ    -288
_LVOUnlockIBase      equ    -420
_LVOViewAddress      equ    -294
_LVOViewPortAddress  equ    -300
_LVOWBenchToBack     equ    -336
_LVOWBenchToFront    equ    -342
_LVOWindowLimits     equ    -318
_LVOWindowToBack     equ    -306
_LVOWindowToFront    equ    -312
```

A 2.4 Graphics-Library

Name zum Öffnen: `graphics.library`
Basis-Adresse: `_GfxBase`

```
_LVOAddAnimOb        equ    -156
_LVOAddBob           equ    -96
_LVOAddFont          equ    -480
_LVOAddVSprite       equ    -102
_LVOAllocRaster      equ    -492
_LVOAndRectRegion    equ    -504
_LVOAnimate          equ    -162
_LVOAreaDraw         equ    -258
_LVOAreaEnd          equ    -264
_LVOAreaMove         equ    -252
_LVOAskFont          equ    -474
```

```
_LVOAskSoftStyle        equ     -84
_LVOBltBitMap           equ     -30
_LVOBltBitMapRastPort   equ     -606
_LVOBltClear            equ     -300
_LVOBltPattern          equ     -312
_LVOBltTemplate         equ     -36
_LVOCBump               equ     -366
_LVOChangeSprite        equ     -420
_LVOClearEOL            equ     -42
_LVOClearRegion         equ     -528
_LVOClearScreen         equ     -48
_LVOClipBlit            equ     -552
_LVOCloseFont           equ     -78
_LVOCMove               equ     -372
_LVOCopySBitMap         equ     -450
_LVOCWait               equ     -378
_LVODisownBlitter       equ     -462
_LVODisposeRegion       equ     -534
_LVODoCollision         equ     -108
_LVODraw                equ     -246
_LVODrawGList           equ     -114
_LVOFlood               equ     -330
_LVOFreeColorMap        equ     -576
_LVOFreeCopList         equ     -546
_LVOFreeCprList         equ     -564
_LVOFreeGBuffers        equ     -600
_LVOFreeRaster          equ     -498
_LVOFreeSprite          equ     -414
_LVOFreeVPortCopLists   equ     -540
_LVOGelsFuncE           equ     -180
_LVOGelsFuncF           equ     -186
_LVOGetColorMap         equ     -570
_LVOGetGBuffers         equ     -168
_LVOGetRGB4             equ     -582
_LVOGetSprite           equ     -408
_LVOInitArea            equ     -282
_LVOInitBitMap          equ     -390
_LVOInitGels            equ     -120
_LVOInitGMasks          equ     -174
_LVOInitMasks           equ     -126
_LVOInitRastPort        equ     -198
_LVOInitTmpRas          equ     -468
_LVOInitView            equ     -360
_LVOInitVPort           equ     -204
_LVOLoadRGB4            equ     -192
_LVOLoadView            equ     -222
_LVOLockLayerRom        equ     -432
_LVOMakeVPort           equ     -216
_LVOMove                equ     -240
_LVOMoveSprite          equ     -426
_LVOMrgCop              equ     -210
_LVONewRegion           equ     -516
```

```
_LVONotRegion          equ     -522
_LVOOpenFont           equ     -72
_LVOOrRectRegion       equ     -510
_LVOOwnBlitter         equ     -456
_LVOPolyDraw           equ     -336
_LVOQBlit              equ     -276
_LVOQBSBlit            equ     -294
_LVOReadPixel          equ     -318
_LVORectFill           equ     -306
_LVORemFont            equ     -486
_LVORemIBob            equ     -132
_LVORemVSprite         equ     -138
_LVOScrollRaster       equ     -396
_LVOScrollVPort        equ     -588
_LVOSetAPen            equ     -342
_LVOSetBPen            equ     -348
_LVOSetCollision       equ     -144
_LVOSetDrMd            equ     -354
_LVOSetFont            equ     -66
_LVOSetRast            equ     -234
_LVOSetRGB4            equ     -288
_LVOSetSoftStyle       equ     -90
_LVOSortGList          equ     -150
_LVOSyncSBitMap        equ     -444
_LVOText               equ     -60
_LVOTextLength         equ     -54
_LVOUCopperListInit    equ     -594
_LVOUnlockLayerRom     equ     -438
_LVOVBeamPos           equ     -384
_LVOWaitBlit           equ     -228
_LVOWaitBOVP           equ     -402
_LVOWaitTOF            equ     -270
_LVOWritePixel         equ     -324
_LVOXorRectRegion      equ     -558
```

A 2.5 Icon-Library

Name zum Öffnen: `icon.library`
Basis-Adresse: `_IconBase`

```
_LVOAddFreeList        equ     -72
_LVOAllocWBObject      equ     -66
_LVOBumpRevision       equ     -108
_LVOFindToolType       equ     -96
_LVOFreeDiskObject     equ     -90
_LVOFreeFreeList       equ     -54
_LVOFreeWBObject       equ     -60
_LVOGetDiskObject      equ     -78
_LVOGetIcon            equ     -42
_LVOGetWBObject        equ     -30
```

```
_LVOMatchToolValue      equ     -102
_LVOPutDiskObject       equ     -84
_LVOPutIcon             equ     -48
_LVOPutWBObject         equ     -36
```

A 2.6 Die Mathematik-Libraries

Name zum Öffnen: `mathffp.library`
Basis-Adresse: `_MathBase`

```
_LVOSPAbs               equ     -54
_LVOSPAdd               equ     -66
_LVOSPCmp               equ     -42
_LVOSPDiv               equ     -84
_LVOSPFix               equ     -30
_LVOSPFlt               equ     -36
_LVOSPMul               equ     -78
_LVOSPNeg               equ     -60
_LVOSPSub               equ     -72
_LVOSPTst               equ     -48
```

Name zum Öffnen: `mathieeedoubbas.library`
Basis-Adresse: `_MathIeeeDoubBasBase`

```
_LVOIEEEDPAbs           equ     -54
_LVOIEEEDPAdd           equ     -66
_LVOIEEEDPCmp           equ     -42
_LVOIEEEDPDiv           equ     -84
_LVOIEEEDPFix           equ     -30
_LVOIEEEDPFlt           equ     -36
_LVOIEEEDPMul           equ     -78
_LVOIEEEDPNeg           equ     -60
_LVOIEEEDPSub           equ     -72
_LVOIEEEDPTst           equ     -48
```

Name zum Öffnen: `mathtrans.library`
Basis-Adresse: `_MathTransBase`

```
_LVOSPAcos              equ     -120
_LVOSPAsin              equ     -114
_LVOSPAtan              equ     -30
_LVOSPCos               equ     -42
_LVOSPCosh              equ     -66
_LVOSPExp               equ     -78
_LVOSPFieee             equ     -108
_LVOSPLog               equ     -84
```

```
_LVOSPLog10         equ     -126
_LVOSPPow           equ     -90
_LVOSPSin           equ     -36
_LVOSPSincos        equ     -54
_LVOSPSinh          equ     -60
_LVOSPSqrt          equ     -96
_LVOSPTan           equ     -48
_LVOSPTanh          equ     -72
_LVOSPTieee         equ     -102
```

A 2.7 Sonstige (Diskfonts und Translator)

Diskfont-Library

Name zum Öffnen: `diskfont.library`
Basis-Adresse: `DiskfontBase`

```
_LVOAvailFonts      equ     -36
_LVOOpenDiskFont    equ     -30
```

Translator-Library

Name zum Öffnen: `translator.library`
Basis-Adresse: `TranslatorBase`

```
_LVOTranslate       equ     -30
```

Anhang A3: Die wichtigsten Funktionen und ihre Parameter

A 3.1 Exec

	A0	A1	A2	A3	D0	D1
AbortIO		IORequest				
AddDevice		Device				
AddHead	List	Node				
AddIntServer		Interrupt			IntNumber	
AddLibrary		Library				
AddPort		Port				
AddResource		Resource				
AddTail		List	Node			
AddTask		Task	initPC	finalPC		
AllocAbs		locatIOn			Size	
Allocate	freeList				Size	
AllocEntry	Entry					
AllocMem					Size	Request
AllocSignal					SignalNum	
AllocTrap					TrapNum	
AvailMem						Request
Cause		Interrupt				
CheckIO		IORequest				
CloseDevice		IORequest				
CloseLibrary		Library				
Deallocate		freeList	memoryBlock		Size	
Disable						
DoIO		IORequest				
Enable						
Enqueue	List	Node				
FindName	List	name				
FindPort		name				

	A0	A1	A2	A3	D0	D1
FindTask		name				
Forbid						
FreeEntry	Entry					
FreeMem		MemoryBlock			Size	
FreeSignal					SignalNum	
FreeTrap					TrapNum	
GetMsg	Port					
Insert	List	Node	pred			
OldOpenLibrary		LibName				
OpenDevice	devName	IORequest			unit	Flags
OpenResource	resName				Version	
Permit						
PutMsg	Port	message				
RemDevice		Device				
RemHead	List					
RemIntServer		Interrupt			IntNumber	
RemLibrary		Library				
Remove		Node				
RemPort		Port				
RemResource		Resource				
RemTail	List					
RemTask		Task				
ReplyMsg		Message				
SendIO		IORequest				
SetExcept					NewSig	SignalSet
SetIntVector		Interrupt			IntNum	Interrupt
SetSignal					NewSig	SignalSet
SetSR					NewSR	Mask
SetTaskPri		Task			Priority	
Signal		Task			SignalSet	
SumLibrary		Library				
SuperState						
UserState					SysStack	
Wait					SignalSet	
WaitIO		IORequest				
WaitPort	Port					

A 3.2 DOS

Funktion/Reg.	D1	D2	D3	D4
Close	File			
CreateDir	Name			
CreateProc	Name	Prior.	SegList	StackSize
CurrentDir	Lock			
DateStamp	Date			
Delay	Timeout			
DeleteFil	Name			
DeviceProc	Name			
DupLock	Lock			
Examine	Lock	FileInfoBLock		
Execute	String	File	File	
Exit	ReturnCode			
ExNext	Lock	FileInfoBLock		
Info	Lock	ParameterBLock		
Input				
IoErr				
IsInteractive	File			
LoadSeg	FileName			
Lock	Name	Type		
Open	Name	AccessMode		
Output				
ParentDir	Lock			
Read	File	Buffer	Len	
ReName	OldName	NewName		
Seek	File	Position	Offset	
SetComment	Name	Comment		
SetProtection	Name	Mask		
UnLoadSeg	Segment			
UnLock	Lock			
WaitForChar	File	Timeout		
Write	File	Buffer	Len	

A 3.3 Intuition

AddGadget	AddPtr/Gadget/Position A0/A1/D0
AllocRemember	RememberKey/Sizef/Flags A0/D0/D1
AlohaWorkbench	wbport A0
AutoRequest	Window/Body/PText/NText/PFlag/NFlag/W/H
	A0 /A1 /A2 /A3 /D0 /D1 /D2/D3
BeginRefresh	Window A0

BuildSysRequest	Window/Body/PosText/NegText/Flags/W/H A0 /A1 /A2 /A3 /D0 /D1/D2
ClearDMRequest	Window A0
ClearMenuStrip	Window A0
ClearPointer	Window A0
CloseScreen	Screen A0
CloseWindow	Window A0
CloseWorkBench	
CurrentTime	Seconds/Micros A0/A1
DisplayAlert	AlertNumber/String/Height D0/A0/D1
DisplayBeep	Screen A0
DoubleClick	sseconds/smicros/cseconds/cmicros D0/D1/D2/D3
DrawBorder	RPort/Border/LeftOffset/TopOffset A0/A1/D0/D1
DrawImage	RPort/Image/LeftOffset/TopOffset A0/A1/D0/D1
EndRefresh	Window/Complete A0/D0
EndRequest	requester/Window A0/A1
FreeRemember	RememberKey/ReallyForget A0/D0
FreeSysRequest	Window A0
GetDefPrefs	Preferences/Size A0/D0
GetPrefs	Preferences/Size A0/D0
InitRequester	req A0
IntuiTextLength	itext A0
Intuition	ievent A0
ItemAddress	MenuStrip/MenuNumber A0/D0
MakeScreen	Screen A0
ModifyIDCMP	Window/Flags A0/D0
ModifyProp	Gadget/Ptr/Req/Flags/HPos/VPos/HBody/VBody A0 /A1 /A2 /D0 /D1 /D2 /D3 /D4
MoveScreen	Screen/dx/dy A0/D0/D1
MoveWindow	Window/dx/dy A0/D0/D1
OffGadget	Gadget/Ptr/Req A0/A1/A2
OffMenu	Window/MenuNumber A0/D0
OnGadget	Gadget/Ptr/Req A0/A1/A2
OnMenu	Window/MenuNumber A0/D0
OpenIntuition	
OpenScreen	OSargs A0
OpenWindow	OWargs A0
OpenWorkBench	
PrintIText	rp/itext/left/top A0/A1/D0/D1
RefreshGadgets	Gadgets/Ptr/Req A0/A1/A2
RemakeDisplay	
RemoveGadget	RemPtr/Gadget A0/A1
ReportMouse	Window/Boolean A0/D0

Request	Requester/Window A0/A1
RethinkDisplay	
ScreenToBack	Screen A0
ScreenToFront	Screen A0
SetDMRequest	Window/req A0/A1
SetMenuStrip	Window/Menu A0/A1
SetPointer	Window/Pointer/Height/Width/Xoffset/Yoffset A0 /A1 /D0 /D1 /D2 /D3
SetPrefs	Preferences/Size/flag A0/D0/D1
SetWindowTitles	Window/Windowtitle/Screentitle A0/A1/A2
ShowTitle	Screen/ShowIt A0/D0
SizeWindow	Window/dx/dy A0/D0/D1
ViewAddress	
ViewPortAddress	Window A0
WBenchToBack	
WBenchToFront	
WindowLimits	Window/minwidth/minheight/maxwidth/maxheight A0 /D0 /D1 /D2 /D3
WindowToBack	Window A0
WindowToFront	Window A0

A 3.4 Graphics

AddAnimOb	obj/animationKey/rastPort A0/A1/A2
AddBob	bob/rastPort A0/A1
AddFont	textFont A1
AddVSprite	vSprite/rastPort A0/A1
AllocRaster	width/height D0/D1
AndRectRegion	rgn/rect A0/A1
Animate	animationKey/rastPort A0/A1
AreaDraw	rastPort/x/y A1/D0/D1
AreaEnd	rastPort A1
AreaMove	rastPort/x/y A1/D0/D1
AskFont	rastPort/textAttr A1/A0
AskSoftStyle	rastPort A1
BltBitMap	srcBitMap/srcX/srcY/destBitMap/destX/destY A0 /D0 /D1 /A1 /D2 /D3 sizeX/sizeY/minterm/mask/tempA D4 /D5 /D6 /D7 /A2
BltBitMapRastPort	srcbm/srcx/srcy/destrp/destX/destY A0 /D0 /D1 /A1 /D2 /D3 sizeX/sizeY/minterm D4 /D5 /D6

BltClear	memory/size/flags A1/D0/D1
BltPattern	rastPort/ras/xl/yl/maxX/maxY/fillBytes
	A1 /A0 /D0/D1/D2/D3 /D4
BltTemplate	src/srcX/srcMod/destRastPort/destX/destY/sizeX/sizeY
	A0/D0 /D1 /A1 /D2 /D3 /D4 /D5
CBump	copperList A1
ChangeSprite	vp/simplesprite/data A0/A1/A2
ClearEOL	rastPort A1
ClearRegion	rgn A0
ClearScreen	rastPort A1
ClipBlit	src/srcX/srcY/destrp/destX/destY/sizeX/sizeY/minterm
	A0/D0 /D1 /A1 /D2 /D3 /D4 /D5 /D6
CloseFont	textFont A1
CMove	copperList/destination/data A1/D0/D1
CopySBitMap	l1/l2 A0/A1
CWait	copperList/x/y A1/D0/D1
DisownBlitter	
DisposeRegion	rgn A0
DoCollision	rasPort A1
Draw	rastPort/x/y A1/D0/D1
DrawGList	rastPort/viewPort A1/A0
Flood	rastPort/mode/x/y A1/D2/D0/D1
FreeColorMap	colormap A0
FreeCopList	coplist A0
FreeCprList	cprlist A0
FreeGBuffers	animationObj/rastPort/doubleBuffer A0/A1/D0
FreeRaster	planeptr/width/height A0/D0/D1
FreeSprite	num D0
FreeVPortCopLists	viewport A0
GelsFuncE	
GelsFuncF	
GetColorMap	entries D0
GetGBuffers	animationObj/rastPort/doubleBuffer A0/A1/D0
GetRGB4	colormap/entry A0/D0
GetSprite	simplesprite/num A0/D0
InitArea	areaInfo/vectorTable/vectorTableSize A0/A1/D0
InitBitMap	bitMap/depth/width/height A0/D0/D1/D2
InitGels	dummyHead/dummyTail/GelsInfo A0/A1/A2
InitGMasks	animationObj A0
InitMasks	vSprite A0
InitRastPort	rastPort A1
InitTmpRas	tmpras/buff/size A0/A1/D0
InitView	view A1

InitVPort	viewPort A0
LoadRGB4	viewPort/colors/count A0/A1/D0
LoadView	view A1
LockLayerRom	layer A5
MakeVPort	view/viewPort A0/A1
Move	rastPort/x/y A1/D0/D1
MoveSprite	viewport/simplesprite/x/y A0/A1/D0/D1
MrgCop	view A1
NewRegion	
NotRegion	rgn A0
OpenFont	textAttr A0
OrRectRegion	rgn/rect A0/A1
OwnBlitter	
PolyDraw	rastPort/count/polyTable A1/D0/A0
QBlit	blit A1
QBSBlit	blit A1
ReadPixel	rastPort/x/y A1/D0/D1
RectFill	rastPort/xl/yl/xu/yu A1/D0/D1/D2/D3
RemFont	textFont A1
RemIBob	bob/rastPort/viewPort A0/A1/A2
RemVSprite	vSprite A0
ScrollRaster	rastPort/dX/dY/minx/miny/maxx/maxy A1 /D0/D1/D2 /D3 /D4 /D5
ScrollVPort	vp A0
SetAPen	rastPort/pen A1/D0
SetBPen	rastPort/pen A1/D0
SetCollision	type/routine/gelsInfo D0/A0/A1
SetDrMd	rastPort/drawMode A1/D0
SetFont	RastPortID/textFont A1/A0
SetRast	rastPort/color A1/D0
SetRGB4	viewPort/index/r/g/b A0/D0/D1/D2/D3
SetSoftStyle	rastPort/style/enable A1/D0/D1
SortGList	rastPort A1
SyncSBitMap	l A0
Text	RastPort/string/count A1/A0/D0
TextLength	RastPort/string/count A1/A0/D0
UCopperListInit	copperlist/num A0/D0
UnlockLayerRom	layer A5
VBeamPos	
WaitBlit	
WaitBOVP	viewport A0
WaitTOF	

WritePixel	rastPort/x/y A1/D0/D1
XorRectRegion	rgn/rect A0/A1

A 3.5 Layers (li steht für layer info)

BeginUpdate	Layer A0
BehindLayer	li/Layer A0/A1
CreateBehindLayer	li/bm/x0/y0/x1/y1/flags/bm2 A0/A1/D0/D1/D2/D3/D4/A2
CreateUpfrontLayer	li/bm/x0/y0/x1/y1/flags/bm2 A0/A1/D0/D1/D2/D3/D4/A2
DeleteLayer	li/Layer A0/A1
DisposeLayerInfo	li A0
EndUpdate	Layer/flag A0/D0
FattenLayerInfo	li A0
InitLayers	li A0
LockLayer	li/Layer A0/A1
LockLayerInfo	li A0
LockLayers	li A0
MoveLayer	li/Layer/dx/dy A0/A1/D0/D1
MoveLayerInFrontOf	Layer/Layer A0/A1
NewLayerInfo	
ScrollLayer	li/Layer/dx/dy A0/A1/D0/D1
SizeLayer	li/Layer/dx/dy A0/A1/D0/D1
SwapBitsRastPortClip-Rect	rp/cr A0/A1
ThinLayerInfo	li A0
UnlockLayer	Layer A0
UnlockLayerInfo	li A0
UnlockLayers	li A0
UpfrontLayer	li/Layer A0/A1
WhichLayer	li/x/y A0/D0/D1

Anhang A4: Datentypen, Strukturen, Offset-Tabellen, Konstanten

Dieser Anhang ist so aufgebaut, daß Sie Teile davon (oder alles) einfach zu Include-Files umbauen können. Da ich Sie nicht zwingen wollte, immer die DC-Lösung oder die Offset-Lösung zu verwenden, wenn ich die eine oder die andere für richtig halte, folgender Kompromiß: Alle Strukturen sind mit DS.x aufgebaut. In der ersten Spalte steht jedoch das Offset. Die letzte Zeile gibt die Größe mittels einer EQU-Direktive an.

Die Offsets in der ersten Spalte sind immer in hex notiert. Für die Operatoren gilt hex nur wie üblich ($ vor der Zahl). Diese Zeile ist ein Beispiel:

```
1A              nw_Title        ds.l    1
```

Daraus können Sie machen:

```
        dc.l  Mein_Titel
```

oder (xx_SIZE ist immer als EQU definiert):

```
NewWindow  ds.b    nw_SIZE
nw_Title   equ     $1A
           lea     NewWindow,a0
           move.l  #Mein_Titel,nw_Title(a0)
```

Die zu einer Struktur gehörenden Konstanten folgen dieser meistens direkt in Form von EQU-Direktiven. Benutzen zwei Strukturen dieselben Konstanten, stehen sie zwischen den Strukturen. Gegebenenfalls stehen Konstanten auch in anderen Anhängen.

Einzelheiten zum Umgang mit den Strukturen und Offset-Tabellen finden Sie im Kapitel 11. Der Hochpfeil (^) heißt »Zeiger auf« (Adresse von).

A 4.1 Exec

```
;nodes
;-----
ØØØ             LN_SUCC ds.l     1
ØØ4             LN_PRED ds.l     1
ØØ8             LN_TYPE ds.b     1
ØØ9             LN_PRI  ds.b     1
ØØA             LN_NAME ds.l     1
                LN_SIZE equ      $ØØE

NT_UNKNOWN      equ     Ø
NT_TASK equ     1
NT_INTERRUPT    equ     2
NT_DEVICE       equ     3
NT_MSGPORT      equ     4
NT_MESSAGE      equ     5
NT_FREEMSG      equ     6
NT_REPLYMSG     equ     7
NT_RESOURCE     equ     8
NT_LIBRARY      equ     9
NT_MEMORY       equ     1Ø
NT_SOFTINT      equ     11
NT_FONT         equ     12
NT_PROCESS      equ     13
NT_SEMAPHORE    equ     14

;Sys-Basis zählt ab hier
;-----------------------

;lists
;----

ØØØ         LH_HEAD         ds.l    1
ØØ4         LH_TAIL         ds.l    1
ØØ8         LH_TAILPRED     ds.l    1
ØØC         LH_TYPE         ds.b    1
ØØD         LH_pad          ds.b    1
            LH_SIZE         equ     $ØØE

;Message-Port-Struktur
;---------------------

ØØE         MP_FLAGS        ds.b    1        ;Flags
ØØF         MP_SIGBIT       ds.b    1        ;Signal-Bit-Nummer
Ø1Ø         MP_SIGTASK      ds.l    1        ;^Task für den Sig
Ø14         MP_MSGLIST      ds.b    LH_SIZE  ;Message-Liste
            MP_SIZE         equ     $Ø22

;Message-Struktur
ØØE         MN_REPLYPORT    ds.l    1        ;^Reply-Port
Ø12         MN_LENGTH       ds.w    1        ;Len in Bytes
```

```
Ø14          MN_SIZE          equ      $Ø14

MP_SOFTINT       equ  MP_SIGTASK
PF_ACTION        equ  3

PA_SIGNAL        equ  Ø
PA_SOFTINT       equ  1
PA_IGNORE        equ  2

;Libraries
;---------

LIB_VECTSIZE     equ      6
LIB_RESERVED     equ      4
LIB_BASE         equ      $FFFFFFFA
LIB_USERDEF      equ      LIB_BASE-(LIB_RESERVED*LIB_VECTSIZE)
LIB_NONSTD       equ      LIB_USERDEF

ØØE          LIB_FLAGS        ds.b     1    ;Flags siehe unten
ØØF          LIB_pad          ds.b     1    ;
Ø1Ø          LIB_NEGSIZE      ds.w     1    ;Anzahl Bytes vor Lib
Ø12          LIB_POSSIZE      ds.w     1    ;             nach Lib
Ø14          LIB_VERSION      ds.w     1    ;(Haupt)Version
Ø16          LIB_REVISION     ds.w     1    ;Untergruppe Version
Ø18          LIB_IDSTRING     ds.l     1    ;^Name
Ø1C          LIB_SUM          ds.l     1    ;Checksum
Ø2Ø          LIB_OPENCNT      ds.w     1    ;Aktuelle Opens
             LIB_SIZE         equ      $Ø22

LIBB_SUMMING     equ      Ø       ;Checksumme wird gerechnet
LIBF_SUMMING     equ      1       ;
LIBB_CHANGED     equ      1       ;Lib wurde geändert
LIBF_CHANGED     equ      2       ;
LIBB_SUMUSED     equ      2       ;1 wenn Checksum-Problem
LIBF_SUMUSED     equ      4
LIBB_DELEXP      equ      3
LIBF_DELEXP      equ      8

;Semaphore Message Port
;----------------------
Ø22         SM_BIDS       ds.w    1          ;Anzahl Lock-Bits
            SM_SIZE       equ     $Ø24

;Unions
;------
SM_LOCKMSG       equ      MP_SIGTASK         ;siehe dort

;Device-Struktur
;----------------
;************ Wie Library mit DD_xxxx **********
DD_SIZE    equ   $22
```

```
Ø22         UNIT_FLAGS      ds.b    1
Ø23         UNIT_pad        ds.b    1
Ø24         UNIT_OPENCNT    ds.w    1
            UNIT_SIZE       equ     $Ø22

 UNITB_ACTIVE    equu    Ø

;INTERRUPTS
;----------

ØØE         IS_DATA         ds.l    1
Ø12         IS_CODE         ds.l    1
            IS_SIZE         equ     $Ø16

ØØØ         IV_DATA         ds.l    1
ØØ4         IV_CODE         ds.l    1
ØØ8         IV_NODE         ds.l    1
            IV_SIZE         equ     $ØØC

SB_SAR          equ     15
SF_SAR          equ     $8ØØØ
SB_TQE          equ     14
SF_TQE          equ     $4ØØØ
SB_SINT         equ     13
SF_SINT         equ     $2ØØØ

SH_PAD          equ     SH_SIZE

SIH_PRIMASK     equ     $ØFØ
SIH_QUEUES      equ     5

Statische System-Variable
;------------------------

Ø22         SoftVer         ds.w    1   ;Kickstart-Version
Ø24         LowMemChkSum    ds.w    1   ;Checksumme Trap-Vekt.
Ø26         ChkBase         ds.l    1   ;Sys-Basis (Komplement)
Ø2A         ColdCapture     ds.l    1   ;^Cold-Start
Ø2E         CoolCapture     ds.l    1   ;^Cool-Start
Ø32         WarmCapture     ds.l    1   ;^Warm-Start
Ø36         SysStkUpper     ds.l    1   ;^Sys-Stack (oben)
Ø3A         SysStkLower     ds.l    1   ;^Sys-Stack (Top)
Ø3E         MaxLocMem       ds.l    1   ;^akt. max. Memory
Ø42         DebugEntry      ds.l    1   ;^Debugger
Ø46         DebugData       ds.l    1   ;^Debugger Data-Segment
Ø4A         AlertData       ds.l    1   ;^Alerts Data
Ø4E         RsvdExt         ds.l    1   ;reserviert
Ø52         ChkSum          ds.w    1   ;Checksumme bis hierher
;Für Interrupt-Quellen:
Ø54         IntVects        equ     $Ø54
Ø54         IVTBE           ds.b    IV_SIZE
Ø6Ø         IVDSKBLK        ds.b    IV_SIZE
```

```
Ø6C         IVSOFTINT      ds.b    IV_SIZE
Ø78         IVPORTS        ds.b    IV_SIZE
Ø84         IVCOPER        ds.b    IV_SIZE
Ø9Ø         IVVERTB        ds.b    IV_SIZE
Ø9C         IVBLIT         ds.b    IV_SIZE
ØA8         IVAUDØ         ds.b    IV_SIZE
ØB4         IVAUD1         ds.b    IV_SIZE
ØCØ         IVAUD2         ds.b    IV_SIZE
ØCC         IVAUD3         ds.b    IV_SIZE
ØD8         IVRBF          ds.b    IV_SIZE
ØE4         IVDSKSYNC      ds.b    IV_SIZE
ØFØ         IVEXTER        ds.b    IV_SIZE
ØFC         IVINTEN        ds.b    IV_SIZE
1Ø8         IVNMI          ds.b    IV_SIZE
;Dynamische System-Variable
114         ThisTask       ds.l    1    ;^aktueller Task
118         IdleCount      ds.l    1    ;Warte-Zähler
11C         DispCount      ds.l    1    ;Dispatch-Zähler
12Ø         Quantum        ds.w    1    ;Zeit-Quantum
122         Elapsed        ds.w    1    ;davon verbraucht
124         SysFlags       ds.w    1    ;diverse
126         IDNestCnt      ds.b    1    ;Inter.-Disable-Tiefe
127         TDNestCnt      ds.b    1    ;Task-Disable-Tiefe

128         AttnFlags      ds.w    1    ;Merker-Flags
12A         AttnResched    ds.w    1    ;
12C         ResModules     ds.l    1    ;^residente Module

13Ø         TaskTrapCode   ds.l    1    ;^Default Trap-Routine
134         TaskExceptCode ds.l    1    ;^Dfault Exception-Rout.
138         TaskExitCode   ds.l    1    ;^Default Exit-Routine
13C         TaskSigAlloc   ds.l    1    ;Default Signal-Maske
14Ø         TaskTrapAlloc  ds.w    1    ;Default Trap-Maske
;List Headers
142         MemList        ds.b   LH_SIZE
15Ø         ResourceList   ds.b   LH_SIZE
15E         DeviceList     ds.b   LH_SIZE
16C         IntrList       ds.b   LH_SIZE
17A         LibList        ds.b   LH_SIZE
188         PortList       ds.b   LH_SIZE
196         TaskReady      ds.b   LH_SIZE
1A4         TaskWait       ds.b   LH_SIZE

1B2         SoftInts       ds.b   5*SH_SIZE
2Ø2         LastAlert      ds.b   16
212         ExecBaseReserved ds.l   1

SYSBASESIZE      equ      $216
;attention flags:
AFB_68Ø1Ø        equ      Ø      ; (auch 68Ø2Ø)
AFB_68Ø2Ø        equ      1
AFB_68881        equ      4
```

```
AFB_PAL         equ     8       ;PAL/NTSC
AFB_50HZ        equ     9       ;Clock-Rate

UNITF_ACTIVE    equ     1
UNITB_INTASK    equ     1
UNITF_INTASK    equ     2

;memory
;------
00E         ML_NUMENTRIES   ds.w    1
            ML_ME           equ     16
            ML_SIZE         equ     16

000         ME_REQS         ds.w    0
000         ME_ADDR         ds.l    1
004         ME_LENGTH       ds.l    1
            ME_SIZE         equ     8
MEMB_PUBLIC     equ     0
MEMF_PUBLIC     equ     1

MEMB_CHIP       equ     1
MEMF_CHIP       equ     2

MEMB_FAST       equ     2
MEMF_FAST       equ     4

MEMB_CLEAR      equ     16
MEMF_CLEAR      equ     $10000

MEMB_LARGEST    equ     17
MEMF_LARGEST    equ     $20000

MEM_BLOCKSIZE   equ 8
MEM_BLOCKMASK   equ (MEM_BLOCKSIZE-1)

00E         MH_ATTRIBUTES   ds.w    1
010         MH_FIRST        ds.l    1
014         MH_LOWER        ds.l    1
018         MH_UPPER        ds.l    1
01C         MH_FREE         ds.l    1
            MH_SIZE         equ     $020

000         MC_NEXT         ds.l    1
004         MC_BYTES        ds.l    1
008         MC_SIZE         ds.l    1

;Task Control-Struktur
;---------------------

00E         TC_FLAGS        ds.b    1   ;Flags
00F         TC_STATE        ds.b    1   ;Status
010         TC_IDNESTCNT    ds.b    1   ;Interr. Disable-Tiefe
```

```
Ø11     TC_TDNESTCNT    ds.b    1       ;Task-Disable-Tiefe
Ø12     TC_SIGALLOC     ds.l    1       ;zugewiesene Signals
Ø16     TC_SIGWAIT      ds.l    1       ;auf die gewartet wird
Ø1A     TC_SIGRECVD     ds.l    1       ;die da sind
Ø1E     TC_SIGEXCEPT    ds.l    1       ;die als Exception gelten
Ø22     TC_TRAPALLOC    ds.w    1       ;zugewiesene Traps
Ø24     TC_TRAPABLE     ds.w    1       ;Traps enabled
Ø26     TC_EXCEPTDATA   ds.l    1       ;^Exception-Daten
Ø2A     TC_EXCEPTCODE   ds.l    1       ;^Exception-Code
Ø2E     TC_TRAPDATA     ds.l    1       ;^Trap-Data
Ø32     TC_TRAPCODE     ds.l    1       ;^Trap-Code
Ø36     TC_SPREG        ds.l    1       ;Stack Pointer
Ø3A     TC_SPLOWER      ds.l    1       ;^Stackframe unten
Ø3E     TC_SPUPPER      ds.l    1       ;^           oben (+2)
Ø42     TC_SWITCH       ds.l    1       ;^Task, der CPU verliert
Ø46     TC_LAUNCH       ds.l    1       ;^Task, der dann kommt
Ø4A     TC_MEMENTRY     ds.b    LH_SIZE ;Speicherbelegung
Ø58     TC_Userdata     ds.l    1       ;^User-Daten
        TC_SIZE         equ     $Ø5C

TB_PROCTIME     equ     Ø
TF_PROCTIME     equ     1
TB_STACKCHK     equ     4
TF_STACKCHK     equ     16
TB_EXCEPT       equ     5
TF_EXCEPT       equ     32
TB_SWITCH       equ     6
TF_SWITCH       equ     64
TB_LAUNCH       equ     7
TF_LAUNCH       equ     128

TS_INVALID      equ     Ø
TS_ADDED        equ     TS_INVALID+1
TS_RUN          equ     TS_ADDED+1
TS_READY        equ     TS_RUN+1
TS_WAIT         equ     TS_READY+1
TS_EXCEPT       equ     TS_WAIT+1
TS_REMOVED      equ     TS_EXCEPT+1

SIGF_ABORT      equ     $ØØØ1
SIGF_CHILD      equ     $ØØØ2
SIGF_BLIT       equ     $ØØ1Ø
SIGF_DOS        equ     $Ø1ØØ

SIGB_ABORT      equ     Ø
SIGB_CHILD      equ     1
SIGB_BLIT       equ     4
SIGB_DOS        equ     8

SYS_SIGALLOC    equ     $ØFFFF
SYS_TRAPALLOC   equ     $Ø8ØØØ
```

```
;  IO-Request-Struktur
;---------------------

Ø14         IO_DEVICE      ds.l    1    ;^Device-Struktur
Ø18         IO_UNIT        ds.l    1    ;^Unit (des Drivers)
Ø1C         IO_COMMAND     ds.w    1    ;^Device Command
Ø1E         IO_FLAGS       ds.b    1    ;Flags
Ø1F         IO_ERROR       ds.b    1    ;oder Warning-Code
            IO_SIZE        equ     $Ø2Ø
;IO-Extension
Ø2Ø         IO_ACTUAL      ds.l    1    ;Übertragene Bytes
Ø24         IO_LENGTH      ds.l    1    ;Gesamt Bytes
Ø28         IO_DATA        ds.l    1    ;^Daten
Ø2C         IO_OFFSET      ds.l    1    ;Offset wenn Seeking
            IOSTD_SIZE     equ     $3Ø

IOB_QUICK       equ     Ø
IOF_QUICK       equ     1
```

A 4.2 DOS

```
;File-Zugriff
;------------
MODE_READWRITE    equ    1ØØ4           ;nur ab V 1.2
MODE_OLDFILE      equ    1ØØ5
MODE_READONLY     equ    MODE_OLDFILE
MODE_NEWFILE      equ    1ØØ6

;SEEK-Funktion, relative Positionen
;-------------
OFFSET_BEGINNING  equ    -1             ;Beginn des Files
OFFSET_BEGINING   equ    OFFSET_BEGINNING
OFFSET_CURRENT    equ     Ø             ;relativ zu Ist
OFFSET_END        equ     1             ;Ende des Files

;Trivialitäten
;-------------
BITSPERBYTE       equ    8
BYTESPERLONG      equ    4
BITSPERLONG       equ    32
MAXINT            equ    $7FFFFFFF
MININT            equ    $8ØØØØØØØ

;Arten von Locks
;---------------
SHARED_LOCK       equ    -2             ;andere Tasks dürfen lesen
ACCESS_READ       equ    SHARED_LOCK
EXCLUSIVE_LOCK    equ    -1             ;dürfen nicht
ACCESS_WRITE      equ    EXCLUSSIVE_LOCK
```

```
;DateStamp
;---------
00        ds_Days          ds.l    1    ;Tage seit 1.1.1978
04        ds_Minute        ds.l    1    ;Minuten seit 00 Uhr
08        ds_Tick          ds.l    1    ;Ticks in lfd. Minute
          ds_SIZEOF        equ     $0C

TICKS_PER_SECOND         equ 50         ;1 Tick = 1/50 sec


;FileInfoBlock
;-------------
00      fib_DiskKey      ds.l    1
04      fib_DirEntryType ds.l    1           ;0=File, >0 = Dir.
08      fib_FileName     ds.b    108         ;trotzdem nur max. 30
74      fib_Protection   ds.l    1           ;siehe equ unten
78      fib_EntryType    ds.l    1
7C      fib_Size         ds.l    1           ;Filegroesse
80      fib_NumBlocks    ds.l    1
84      fib_DateStamp    ds.b    ds_SIZEOF ;letzte Aenderung
90      fib_Comment      ds.b    116
        fib_SIZEOF       equ     $104

FIBB_READ       equ     3
FIBF_READ       equ     8
FIBB_WRITE      equ     2
FIBF_WRITE      equ     4

FIBB_EXECUTE    equ     1
FIBF_EXECUTE    equ     2
FIBB_DELETE     equ     0
FIBF_DELETE     equ     1

;InfoData (einer Diskette)
;--------
;In dieser Struktur stecken BCPL-Zeiger, daher muss sie auf
;eine Langwortgrenze justiert sein!
                CNOP              0,4
00              id_NumSoftErrors  ds.l    1
04              id_UnitNumber     ds.l    1
08              id_DiskState      ds.l    1   ;Siehe equ unten
0C              id_NumBlocks      ds.l    1
10              id_NumBlocksUsed  ds.l    1
14              id_BytesPerBlock  ds.l    1
18              id_DiskType       ds.l    1
1C              id_VolumeNode     ds.l    1
20              id_InUse          ds.l    1   ;0, wenn nicht
                id_SIZEOF         equ     $24

ID_WRITE_PROTECTED   equ   80
ID_VALIDATING        equ   81
ID_VALIDATED         equ   82
```

```
ID_NO_DISK_PRESENT    equ    -1
ID_UNREADABLE_DISK    equ    $42414400          ; 'BAD'  geshifted
ID_NOT_REALLY_DOS     equ    $E444F53           ; 'NDOS'
ID_DOS_DISK           equ    $44F5300           ; 'DOS'
ID_KICKSTART_DISK     equ    $B49434B           ; 'KICK'

;Error-Codes
;-----------
ERROR_NO_FREE_STORE            equ  103
ERROR_OBJECT_IN_USE            equ  202
ERROR_OBJECT_EXISTS            equ  203
ERROR_OBJECT_NOT_FOUND         equ  205
ERROR_ACTION_NOT_KNOWN         equ  209
ERROR_INVALID_COMPONENT_NAME   equ  210
ERROR_INVALID_LOCK             equ  211
ERROR_OBJECT_WRONG_TYPE        equ  212
ERROR_DISK_NOT_VALIDATED       equ  213
ERROR_DISK_WRITE_PROTECTED     equ  214
ERROR_RENAME_ACROSS_DEVICES    equ  215
ERROR_DIRECTORY_NOT_EMPTY      equ  216
ERROR_DEVICE_NOT_MOUNTED       equ  218
ERROR_SEEK_ERROR               equ  219
ERROR_COMMENT_TOO_BIG          equ  220
ERROR_DISK_FULL                equ  221
ERROR_DELETE_PROTECTED         equ  222
ERROR_WRITE_PROTECTED          equ  223
ERROR_READ_PROTECTED           equ  224
ERROR_NOT_A_DOS_DISK           equ  225
ERROR_NO_DISK                  equ  226
ERROR_NO_MORE_ENTRIES          equ  232

;empfohlene Return-Codes
;-----------------------
RETURN_OK        equ    0      ;alles bestens
RETURN_WARN      equ    5      ;nur Warnung/Hinweis
RETURN_ERROR     equ   10      ;Fehler
RETURN_FAIL      equ   20      ;Totalschaden

;Break-Codes
;-----------
SIGBREAKB_CTRL_C        equ     12
SIGBREAKF_CTRL_C        equ     $1000
SIGBREAKB_CTRL_D        equ     13
SIGBREAKF_CTRL_D        equ     $2000
SIGBREAKB_CTRL_E        equ     14
SIGBREAKF_CTRL_E        equ     $4000
SIGBREAKB_CTRL_F        equ     15
SIGBREAKF_CTRL_F        equ     $8000
```

A 4.3 Intuition

```
;Die folgenden Konstanten stehen hier nur hilfsweise. Sie
;wurden aus den anderen Anhaengen A4.x uebernommen, um die
;Abhängigkeiten aufzeigen zu koennen.

bm_SIZEOF       EQU     $28                     ;aus gfx
vp_SIZEOF       equ     $28                     ;aus view
rp_SIZEOF       equ     $64                     ;aus rastport
RP_JAM2         equ     1
li_SIZEOF       equ     $66                     ;aus layers
MN_SIZE         equ     $14                     ;aus port
TV_SIZE         equ     8                       ;aus timer

;Ende Hilfe
;------------------------------------------------------------

;-------------------------------------------
;               Menu-Titel
;-------------------------------------------

ØØ              mu_NextMenu     ds.l    1       ;^Nächster Titel
Ø4              mu_LeftEdge     ds.w    1       ;links
Ø6              mu_TopEdge      ds.w    1       ;       oben
Ø8              mu_Width        ds.w    1       ;Breite
ØA              mu_Height       ds.w    1       ;Höhe
ØC              mu_Flags        ds.w    1       ;Bits siehe unten
ØE              mu_MenuName     ds.l    1       ;^Titel-Text
12              mu_FirstItem    ds.l    1       ;^Item-Liste
16              mu_JazzX        ds.w    1       ;intern
18              mu_JazzY        ds.w    1
1A              mu_BeatX        ds.w    1
1C              mu_BeatY        ds.w    1
                mu_SIZEOF       equ     $1E

MENUENABLED     equ $ØØØ1                       ;aktiv
MIDRAWN         equ $Ø1ØØ                       ;gezeichnet

;-------------------------------------------
;               Menu-Items
;-------------------------------------------

ØØ              mi_NextItem      ds.l   1       ;^Nächstes Item
Ø4              mi_LeftEdge      ds.w   1       ;links
Ø6              mi_TopEdge       ds.w   1       ;       oben
Ø8              mi_Width         ds.w   1       ;Breite
ØA              mi_Height        ds.w   1       ;Höhe
ØC              mi_Flags         ds.w   1       ;siehe unten
ØE              mi_MutualExclude ds.l   1       ;je Item 1 Bit
12              mi_ItemFill      ds.l   1       ;^Text oder ^Image
16              mi_SelectFill    ds.l   1       ;siehe unten
1A              mi_Command       ds.b   1       ;Taste
```

```
1B              mi_AdjustToWord  ds.b    1
1C              mi_SubItem       ds.l    1     ;^Subitem-Liste
2Ø              mi_NextSelect    ds.w    1     ;Noch ein Item?
                mi_SIZEOF        equ     $22

CHECKIT          equ $ØØØ1     ;Attribut-Item
ITEMTEXT         equ $ØØØ2     ;Item hat Text sonst Image
COMMSEQ          equ $ØØØ4     ;Item hat Taste
MENUTOGGLE       equ $ØØØ8
ITEMENABLED      equ $ØØ1Ø
HIGHFLAGS        equ $ØØCØ     ;Highligt an, dann:
HIGHIMAGE        equ $ØØØØ     ;   alternativ Image/Text
HIGHCOMP         equ $ØØ4Ø     ;Komplement aller Bits in Item
HIGHBOX          equ $ØØ8Ø     ;Box um die Item-Box
HIGHNONE         equ $ØØCØ     ;Keinerlei Highligting
CHECKED          equ $Ø1ØØ     ;Check-Marke wenn gewählt
ISDRAWN          equ $1ØØØ     ;1 wenn Item auf Schirm
HIGHITEM         equ $2ØØØ     ;1 wenn highligted
MENUTOGGLED      equ $4ØØØ     ;1 wenn toggled

NOMENU           equ  $ØØ1F    ;1/Ø= Menu an/aus
NOITEM           equ  $ØØ3F    ;     Item an/aus
NOSUB            equ  $ØØ1F    ;     Subitems an/aus
MENUNULL         equ  $FFFF    ;Kein Item angewählt
CHECKWIDTH       equ  19       ;Platz für Check-Marke
COMMWIDTH        equ  27       ;Platz für Taste wenn HighRes
LOWCHECKWIDTH    equ  13       ;wenn niedrige Auflösung

;--------------------------------------------
;                Requester
;--------------------------------------------

ØØ              rq_OlderRequest ds.l   1     ;Vorgänger
Ø4              rq_LeftEdge     ds.w   1     ;links
Ø6              rq_TopEdge      ds.w   1     ;       oben
Ø8              rq_Width        ds.w   1     ;Breite
ØA              rq_Height       ds.w   1     ;Höhe
ØC              rq_RelLeft      ds.w   1     ;Wenn Pointer
ØE              rq_RelTop       ds.w   1     ;  Bezugspunkt
1Ø              rq_ReqGadget    ds.l   1     ;^Gadget-Liste
14              rq_ReqBorder    ds.l   1     ;^Border-Struktur
18              rq_ReqText      ds.l   1     ;^Text-Struktur
1C              rq_Flags        ds.w   1     ;Bits siehe unten
1E              rq_BackFill     ds.b   1     ;Hintergrund-Pen
1F              rq_AdjustToWord ds.b   1
2Ø              rq_ReqLayer     ds.l   1     ;^Layer-Struktur
24              rq_ReqPad1      ds.b   32    ;Reserviert
44              rq_ReqBMap      ds.l   1     ;^Custom Bit Map
48              rq_RWindow      ds.l   1     ;Reserviert
4C              rq_ReqPad2      ds.b   36    ;Reserviert
                rq_SIZEOF       equ    $7Ø
```

```
POINTREL        equ $ØØØ1       ;1 wenn relativ zu Mauszeiger
PREDRAWN        equ $ØØØ2       ;1 wenn Custom Bit Map
REQOFFWINDOW    equ $1ØØØ       ;wenn Requ. außerhalb Window
REQACTIVE       equ $2ØØØ       ;Ø/1= Requ. aktiv
SYSREQUEST      equ $4ØØØ       ;nur wenn System-Requester
DEFERREFRESH    equ $8ØØØ       ;

;-------------------------------------------
;               Gadgets
;-------------------------------------------

ØØ              gg_NextGadget   ds.l    1       ;^Nächstes Gadget
Ø4              gg_LeftEdge     ds.w    1       ;links
Ø6              gg_TopEdge      ds.w    1       ;       oben
Ø8              gg_Width        ds.w    1       ;Breite
ØA              gg_Height       ds.w    1       ;Höhe
ØC              gg_Flags        ds.w    1       ;Bits siehe unten
ØE              gg_Activation   ds.w    1       ;Bits siehe unten
1Ø              gg_GadgetType   ds.w    1       ;Bool/Str/Prop
12              gg_GadgetRender ds.l    1       ;^Image oder ^Border
16              gg_SelectRender ds.l    1       ;^Alternative von "
1A              gg_GadgetText   ds.l    1       ;^Text-Struktur
1E              gg_MutualExcludeds.l    1       ;Ohne Wirkung!?
22              gg_SpecialInfo  ds.l    1       ;^Str oder PropInfo
26              gg_GadgetID     ds.w    1       ;beliebige User-ID
28              gg_UserData     ds.l    1       ;^beliebige Daten
2C              gg_SIZEOF       equ     $2C

AUTOFRONTPEN    equ   Ø  ;empfohlene Werte für Auto-Request
AUTOBACKPEN     equ   1  ;siehe IntuitionText
AUTODRAWMODE    equ   1
AUTOLEFTEDGE    equ   6
AUTOTOPEDGE     equ   3
AUTOITEXTFONT   equ   Ø
AUTONEXTTEXT    equ   Ø


GADGHIGHBITS    equ $ØØØ3       ;Kein Highliting oder:
GADGHCOMP       equ $ØØØØ       ;  Komplement aller Bits
GADGHBOX        equ $ØØØ1       ;  Box um Gadget
GADGHIMAGE      equ $ØØØ2       ;  alternatives Image/Border
GADGHNONE       equ $ØØØ3       ;  Kein Highliting

GADGIMAGE       equ $ØØØ4       ;Ø/1=Border/Image
GRELBOTTOM      equ $ØØØ8       ;Ø/1= relativ zu Top/Bottom-Grenze
GRELRIGHT       equ $ØØ1Ø       ;Ø/1= relativ zu links/rechts
GRELWIDTH       equ $ØØ2Ø       ;Absolute/Relative Breite
GRELHEIGHT      equ $ØØ4Ø       ;                  Höhe
SELECTED        equ $ØØ8Ø       ;Vorwahl aus/an
GADGDISABLED    equ $Ø1ØØ       ;für An/Aus-Gadgets
RELVERIFY       equ $ØØØ1       ;Release Verify
GADGIMMEDIATE   equ $ØØØ2       ;Sofort Nachricht wenn
```

```
ENDGADGET       equ $0004     ;Requester vom Schirm
FOLLOWMOUSE     equ $0008     ;Sende Maus-Koordinaten
RIGHTBORDER     equ $0010     ;justieren nach rechts
LEFTBORDER      equ $0020     ;                links
TOPBORDER       equ $0040     ;                oben
BOTTOMBORDER    equ $0080     ;                unten
TOGGLESELECT    equ $0100     ;toggelt Gadget
STRINGCENTER    equ $0200     ;Justiert Text
STRINGRIGHT     equ $0400     ;
LONGINT         equ $0800     ;Erlaube Long Int im String Gadget
ALTKEYMAP       equ $1000     ;Wenn vorhanden und in StringInfo

;Einer dieser Typen muß sein:
BOOLGADGET      equ $0001
GADGET0002      equ $0002
PROPGADGET      equ $0003
STRGADGET       equ $0004

GADGETTYPE      equ $FC00
SYSGADGET       equ $8000     ;System-Gadget (vergibt Intuition)
SCRGADGET       equ $4000     ;wenn im Screen
GZZGADGET       equ $2000     ;wenn im Gimmezerozero-Window
REQGADGET       equ $1000     ;wenn im Requester
SIZING          equ $0010     ;Sys-Typen
WDRAGGING       equ $0020
SDRAGGING       equ $0030
WUPFRONT        equ $0040
SUPFRONT        equ $0050
WDOWNBACK       equ $0060
SDOWNBACK       equ $0070
CLOSE           equ $0080

;   PropInfo (für Proportional-Gadgets)
;---------------------------------------------

00            pi_Flags        ds.w    1      ;siehe unten
02            pi_HorizPot     ds.w    1      ;Horizontal %
04            pi_VertPot      ds.w    1      ;Vertikal %
06            pi_HorizBody    ds.w    1      ;entweder den
08            pi_VertBody     ds.w    1      ;oder den anzeigen
0A            pi_CWidth       ds.w    1      ;Breite Rahmen
0C            pi_CHeight      ds.w    1      ;Höhe
0E            pi_HPotRes      ds.w    1      ;Schrittweite hor.
10            pi_VPotRes      ds.w    1      ;             ver.
12            pi_LeftBorder   ds.w    1      ;Rahmenlage links
14            pi_TopBorder    ds.w    1      ;           oben
16            pi_SIZEOF       equ     $16

AUTOKNOB        equ $0001     ;Autom. Knopf
FREEHORIZ       equ $0002     ;Knopfbewegung horizontal
FREEVERT        equ $0004     ;              vertikal
PROPBORDERLESS  equ $0008     ;Ohne Border
```

```
KNOBHIT         equ $Ø1ØØ       ;1 wenn Knopf berührt wird
KNOBHMIN        equ 6           ;Limits:
KNOBVMIN        equ 4
MAXBODY         equ $FFFF
MAXPOT          equ $FFFF
;     StringInfo (für String-Gadgets)
;-------------------------------------------

ØØ           si_Buffer       ds.l    1  ;^Arbeitspuffer
Ø4           si_UndoBuffer   ds.l    1  ;^Undo-Buffer oder Ø
Ø8           si_BufferPos    ds.w    1  ;Anfangs-Cursor-Position
ØA           si_MaxChars     ds.w    1  ;Puffergröße + 1;
ØC           si_DispPos      ds.w    1  ;Pos. Cursor-Zeichen
ØE           si_UndoPos      ds.w    1  ;Cursor in Undo-Buffer
1Ø           si_NumChars     ds.w    1  ;Zeichen im Puffer
12           si_DispCount    ds.w    1  ;Zeichen sichtbar
14           si_CLeft        ds.w    1  ;Lage des Rahmens
16           si_CTop         ds.w    1  ;
18           si_LayerPtr     ds.l    1  ;^Layer des Gadgets
1C           si_LongInt      ds.l    1  ;Long Int hier
2Ø           si_AltKeyMap    ds.l    1  ;^eigene Keymap
             si_SIZEOF       equ    $24

;-------------------------------------------
;               Intuition Text
;-------------------------------------------

ØØ           it_FrontPen     ds.b    1  ;Vordergrund-Farbe
Ø1           it_BackPen      ds.b    1  ;Hintergrund
Ø2           it_DrawMode     ds.b    1  ;JAM1, JAM2 oder XOR
Ø3           it_AdjustToWord ds.b    1
Ø4           it_LeftEdge     ds.w    1  ;Lage links
Ø6           it_TopEdge      ds.w    1       oben
Ø8           it_ITextFont    ds.l    1  ;^Font-Struktur oder Ø
ØC           it_IText        ds.l    1  ;^Text-String (Ø-term.)
1Ø           it_NextText     ds.l    1  ;^Nächste Struktur o.Ø
             it_SIZEOF       equ    $14

;-------------------------------------------
;               Borders (Polygone)
;-------------------------------------------

ØØ           bd_LeftEdge     ds.w    1  ;Start links
Ø2           bd_TopEdge      ds.w    1  ;      oben
Ø4           bd_FrontPen     ds.b    1  ;Vordergrund-Farbe
Ø5           bd_BackPen      ds.b    1  ;ohne Wirkung
Ø6           bd_DrawMode     ds.b    1  ;JAM1 oder XOR
Ø7           bd_Count        ds.b    1  ;Anzahl Paare
Ø8           bd_XY           ds.l    1  ;^Array mit Paaren
ØC           bd_NextBorder   ds.l    1  ;^Nächstes oder Ø
1Ø           bd_SIZEOF       equ    $1Ø
```

```
;----------------------------------------
;               Images
;----------------------------------------

00          ig_LeftEdge     ds.w    1  ;Lage links
02          ig_TopEdge      ds.w    1  ;     oben
04          ig_Width        ds.w    1  ;     Breite
06          ig_Height       ds.w    1  ;     Höhe
08          ig_Depth        ds.w    1  ;Anzahl Bitplanes
0A          ig_ImageData    ds.l    1  ;^Bitmuster
0E          ig_PlanePick    ds.b    1  ;Genutzte Planes
0F          ig_PlaneOnOff   ds.b    1  ;
10          ig_NextImage    ds.l    1  ;^Nächste Struktur
14          ig_SIZEOF       equ     $14

;----------------------------------------
;            Intuition Message
;----------------------------------------

00          im_ExecMessage  ds.b    MN_SIZE ;reserviert
14          im_Class        ds.l    1  ;Bits wie IDCMP-Flags
18          im_Code         ds.w    1  ;Werte hier
1A          im_Qualifier    ds.w    1  ;für RAW-IO
1C          im_IAddress     ds.l    1  ;Adresse von Objekten
20          im_MouseX       ds.w    1  ;Mouse-Koordinaten
22          im_MouseY       ds.w    1  ;
24          im_Seconds      ds.l    1  ;System-Zeit
28          im_Micros       ds.l    1  ;
2C          im_IDCMPWindow  ds.l    1  ;Adresse des Fensters
30          im_SpecialLink  ds.l    1  ;reserviert
            im_SIZEOF       equ     $34

SIZEVERIFY      equ   $00000001    ;Message wenn Versuch Sizing
NEWSIZE         equ   $00000002    ;Message wenn Sizing fertig
REFRESHWINDOW   equ   $00000004    ;Message wenn Refresh nötig
MOUSEBUTTONS    equ   $00000008    ;Message wenn Mouse-Events
MOUSEMOVE       equ   $00000010    ;
GADGETDOWN      equ   $00000020    ;Message wenn Gadget-Event
GADGETUP        equ   $00000040    ;
REQSET          equ   $00000080    ;Message wenn Requester
MENUPICK        equ   $00000100    ;Message wenn Menu-Event
CLOSEWINDOW     equ   $00000200    ;Message wenn Close_Gadget
RAWKEY          equ   $00000400    ;Message wenn Raw-Key
REQVERIFY       equ   $00000800    ;Warte bevor Requester erlaubt
REQCLEAR        equ   $00001000    ;Message wenn letzter Requ. weg
MENUVERIFY      equ   $00002000    ;Warte bis Menus gezeichnet
NEWPREFS        equ   $00004000    ;Message wenn Prefs. geändert
DISKINSERTED    equ   $00008000    ;Message wenn Diskette
DISKREMOVED     equ   $00010000    ;                rein/raus
WBENCHMESSAGE   equ   $00020000    ;
ACTIVEWINDOW    equ   $00040000    ;
INACTIVEWINDOW  equ   $00080000
```

```
DELTAMOVE       equ   $00100000    ;Mouse-Pos. relativ
VANILLAKEY      equ   $00200000    ;Message wenn Key in Code
INTUITICKS      equ   $00400000    ;Message nach 1/10 Sekunde
LONELYMESSAGE   equ   $80000000
;für Menu-Verify:
MENUHOT         equ   $0001   ;Cancel muß verifiziert werden
MENUCANCEL      equ   $0002   ;Hot Reply cancelt Menu
MENUWAITING     equ   $0003   ;Int. wartet auf Reply

WBENCHOPEN      equ $0001
WBENCHCLOSE     equ $0002

;--------------------------------------------
;               NewWindow
;--------------------------------------------

00              nw_LeftEdge     ds.w    1     ;links
02              nw_TopEdge      ds.w    1     ;      oben
04              nw_Width        ds.w    1     ;Breite
06              nw_Height       ds.w    1     ;Höhe
08              nw_DetailPen    ds.b    1     ;Fein-Stift
09              nw_BlockPen     ds.b    1     ;Grob-Stift
0A              nw_IDCMPFlags   ds.l    1     ;Bits siehe unten
0E              nw_Flags        ds.l    1     ;
12              nw_FirstGadget  ds.l    1     ;^User-Gadgets
16              nw_CheckMark    ds.l    1     ;^User-Checkmark
1A              nw_Title        ds.l    1     ;^Titel-Text
1E              nw_Screen       ds.l    1     ;^Screen
22              nw_BitMap       ds.l    1     ;^User-Bitmap
26              nw_MinWidth     ds.w    1     ;Min. Breite
28              nw_MinHeight    ds.w    1     ;     Höhe
2A              nw_MaxWidth     ds.w    1     ;Max. Breite
2C              nw_MaxHeight    ds.w    1     ;     Höhe
2E              nw_Type ds.w    1             ;Screen-Typ
                nw_SIZE         equ     $30

WINDOWSIZING    equ $0001     ;erlaubte Gadgets
WINDOWDRAG      equ $0002
WINDOWDEPTH     equ $0004
WINDOWCLOSE     equ $0008

SIZEBRIGHT      equ $0010     ;Sizing-Gadget rechts außen
SIZEBBOTTOM     equ $0020     ;                     innen

REFRESHBITS     equ $00C0     ;Eines sollte sein:
SMART_REFRESH   equ $0000     ;
SIMPLE_REFRESH  equ $0040     ;
SUPER_BITMAP    equ $0080     ;
OTHER_REFRESH   equ $00C0     ;

BACKDROP        equ $0100     ;Wenn Backdrop-Window
REPORTMOUSE     equ $0200     ;Message wenn Mouse-Event
```

```
GIMMEZEROZERO   equ $0400       ;Wenn das gewünscht
BORDERLESS      equ $0800       ;wenn kein Rahmen

ACTIVATE        equ $1000       ;Aktiv nach Open
WINDOWACTIVE    equ $2000       ;Message wenn aktiviert
INREQUEST       equ $4000       ;Wd in Request-Modus
MENUSTATE       equ $8000       ;Message wenn Menu

RMBTRAP         equ $00010000 ;Message wenn rechte Maustaste
NOCAREREFRESH   equ $00020000 ;keine Message bei Refresh

WINDOWREFRESH   equ $01000000
WBENCHWINDOW    equ $02000000
WINDOWTICKED    equ $04000000

SUPER_UNUSED    equ $FCFC0000

;-------------------------------------------
;               Window
;-------------------------------------------

00            wd_NextWindow   ds.l    1      ;^nächstes Window
04            wd_LeftEdge     ds.w    1      ;links
06            wd_TopEdge      ds.w    1      ;      oben
08            wd_Width        ds.w    1      ;Breite
0A            wd_Height       ds.w    1      ;Höhe
0C            wd_MouseY       ds.w    1      ;Maus X
0E            wd_MouseX       ds.w    1      ;     Y

10            wd_MinWidth     ds.w    1      ;Min. Breite
12            wd_MinHeight    ds.w    1      ;     Höhe
14            wd_MaxWidth     ds.w    1      ;Max. Breite
16            wd_MaxHeight    ds.w    1      ;     Höhe
18            wd_Flags        ds.l    1      ;Bits siehe oben
1C            wd_MenuStrip    ds.l    1      ;^Menu-Liste
20            wd_Title        ds.l    1      ;^Titel-Text
24            wd_FirstRequest ds.l    1      ;^erster Requ.
28            wd_DMRequest    ds.l    1      ;^DM-Requ.
2C            wd_ReqCount     ds.w    1      ;Anzahl Requ.
2E            wd_WScreen      ds.l    1      ;^Screen
32            wd_RPort        ds.l    1      ;^RastPort
36            wd_BorderLeft   ds.b    1      ;aktuelle Lage von:
37            wd_BorderTop    ds.b    1      ;
38            wd_BorderRight  ds.b    1      ;
39            wd_BorderBottom ds.b    1      ;
3A            wd_BorderRPort  ds.l    1      ;^RastPortG00-Wd.außen
3E            wd_FirstGadget  ds.l    1      ;^Gadget-Liste
42            wd_Parent       ds.l    1      ;in Liste
46            wd_Descendant   ds.l    1      ;
4A            wd_Pointer      ds.l    1      ;^Mauszeiger-Struktur
4E            wd_PtrHeight    ds.b    1      ;            Höhe
4F            wd_PtrWidth     ds.b    1      ;            Breite
```

```
50            wd_XOffset      ds.b    1    ;          Offset
51            wd_YOffset      ds.b    1    ;
52            wd_IDCMPFlags   ds.l    1    ;Bits siehe oben
56            wd_UserPort     ds.l    1    ;^Empfangsport
5A            wd_WindowPort   ds.l    1    ;^Sende-Port
5E            wd_MessageKey   ds.l    1    ;^Int.-Message
62            wd_DetailPen    ds.b    1    ;Fein
63            wd_BlockPen     ds.b    1    ;Grob
64            wd_CheckMark    ds.l    1    ;^User-Checkmark
68            wd_ScreenTitle  ds.l    1    ;^Screen-Titel (Ø)
6C            wd_GZZMouseX    ds.w    1    ;Nur wenn GØØ-Window
6E            wd_GZZMouseY    ds.w    1
7Ø            wd_GZZWidth     ds.w    1
72            wd_GZZHeight    ds.w    1
74            wd_ExtData      ds.l    1    ;Zwei Zeiger für
78            wd_UserData     ds.l    1    ;User
7C            wd_WLayer       ds.l    1    ;^Layer des Wd

;-------------------------------------------
;              NewScreen
;-------------------------------------------

ØØ            ns_LeftEdge     ds.w    1    ;links
Ø2            ns_TopEdge      ds.w    1    ;      oben
Ø4            ns_Width        ds.w    1    ;Breite
Ø6            ns_Height       ds.w    1    ;Höhe
Ø8            ns_Depth        ds.w    1    ;Bitplanes
ØA            ns_DetailPen    ds.b    1    ;Zeichen-
ØB            ns_BlockPen     ds.b    1    ;       Stifte
ØC            ns_ViewModes    ds.w    1    ;Ø, HIRES usw.
ØE            ns_Type         ds.w    1    ;CUSTOM oder Bitmap
1Ø            ns_Font         ds.l    1    ;^Font oder Ø
14            ns_DefaultTitle ds.l    1    ;^Titeltext
18            ns_Gadgets      ds.l    1    ;Immer Ø setzen!!
1C            ns_CustomBitMap ds.l    1    ;^auf eigene Bitmap
              ns_SIZEOF       equ     $2Ø

SCREENTYPE      equ   $ØØØF   ;alle Typen
WBENCHSCREEN    equ   $ØØØ1   ;Workbench
CUSTOMSCREEN    equ   $ØØØF   ;eigener
SHOWTITLE       equ   $ØØ1Ø   ;1 wenn ShowTitle gerufen
BEEPING         equ   $ØØ2Ø   ;1 wenn Beep (Blink)
CUSTOMBITMAP    equ   $ØØ4Ø   ;1 wenn eigene Bitmap

FILENAME_SIZE   equ 3Ø

POINTERSIZE     equ   36

TOPAZ_EIGHTY    equ   8
TOPAZ_SIXTY     equ   9
```

```
;-------------------------------------------
;               Screen
;-------------------------------------------

ØØØ        sc_NextScreen  ds.l    1        ;^Nächster
ØØ4        sc_FirstWindow ds.l    1        ;^erstes Window
ØØ8        sc_LeftEdge    ds.w    1        ;links
ØØA        sc_TopEdge     ds.w    1        ;      oben
ØØC        sc_Width       ds.w    1        ;Breite
ØØE        sc_Height      ds.w    1        ;Höhe
Ø1Ø        sc_MouseY      ds.w    1        ;Maus-Lage
Ø12        sc_MouseX      ds.w    1        ;
Ø14        sc_Flags       ds.w    1        ;Bits siehe oben
Ø16        sc_Title       ds.l    1        ;^Titel-Text
Ø1A        sc_DefaultTitle ds.l   1        ;^Text für Wd ohne
Ø1E        sc_BarHeight   ds.b    1        ;Größe der Bars
Ø1F        sc_BarVBorder  ds.b    1        ;  für Screen
Ø2Ø        sc_BarHBorder  ds.b    1        ;  und alle seine Wd
Ø21        sc_MenuVBorder ds.b    1        ;::
Ø22        sc_MenuHBorder ds.b    1
Ø23        sc_WBorTop     ds.b    1
Ø24        sc_WBorLeft    ds.b    1
Ø25        sc_WBorRight   ds.b    1
Ø26        sc_WBorBottom  ds.b    1
Ø27        sc_AdjustToWord ds.b   1
Ø28        sc_Font        ds.l    1        ;^Default Font
Ø2C        sc_ViewPort    ds.b    $64      ;Display-Art
Ø54        sc_RastPort    ds.b    rp_SIZEOF ;Zeichen-Art
ØB8        sc_BitMap      ds.b    $28      ;ext. Bitmap-Strukt
ØEØ        sc_LayerInfo   ds.b    li_SIZEOF ;Layer-Info
146        sc_FirstGadget ds.l    1        ;^Gadget-Liste
14A        sc_DetailPen   ds.b    1        ;Zeichen-Stifte
14B        sc_BlockPen    ds.b    1        ;
14C        sc_SaveColorØ  ds.w    1        ;für Beep
14E        BarLayer       ds.l    1        ;sc_fehlt wirklich!
152        sc_ExtData     ds.l    1        ;^User-Data
156        sc_UserData    ds.l    1        ;der auch
           sc_SIZEOF      equ     $15A

;-------------------------------------------------
;               Preferences  (Siehe dazu Equates unten)
;-------------------------------------------------

ØØ         pf_FontHeight      ds.b    1
Ø1         pf_PrinterPort     ds.b    1
Ø2         pf_BaudRate        ds.w    1
Ø4         pf_KeyRptSpeed     ds.b    TV_SIZE
ØC         pf_KeyRptDelay     ds.b    TV_SIZE
14         pf_DoubleClick     ds.b    TV_SIZE
1C         pf_PointerMatrix   ds.b    POINTERSIZE*2
64         pf_XOffset         ds.b    1
65         pf_YOffset         ds.b    1
```

```
66          pf_color17          ds.w    1
68          pf_color18          ds.w    1
6A          pf_color19          ds.w    1
6C          pf_PointerTicks     ds.w    1
6E          pf_colorØ           ds.w    1
7Ø          pf_color1           ds.w    1
72          pf_color2           ds.w    1
74          pf_color3           ds.w    1
76          pf_ViewXOffset      ds.b    1
77          pf_ViewYOffset      ds.b    1
78          pf_ViewInitX        ds.w    1
7A          pf_ViewInitY        ds.w    1
7C          EnableCLI           ds.w    1
7E          pf_PrinterType      ds.w    1
8Ø          pf_PrinterFilename  ds.b    FILENAME_SIZE
9E          pf_PrintPitch       ds.w    1
AØ          pf_PrintQuality     ds.w    1
A2          pf_PrintSpacing     ds.w    1
A4          pf_PrintLeftMargin  ds.w    1
A6          pf_PrintRightMargin ds.w    1
A8          pf_PrintImage       ds.w    1
AA          pf_PrintAspect      ds.w    1
AC          pf_PrintShade       ds.w    1
AE          pf_PrintThreshold   ds.w    1
BØ          pf_PaperSize        ds.w    1
B2          pf_PaperLength      ds.w    1
B4          pf_PaperType        ds.w    1
B6          pf_padding          ds.b    5Ø
            pf_SIZEOF           equ     $E8


PARALLEL_PRINTER equ   $ØØ
SERIAL_PRINTER   equ   $Ø1
BAUD_11Ø         equ   $ØØ
BAUD_3ØØ         equ   $Ø1
BAUD_12ØØ        equ   $Ø2
BAUD_24ØØ        equ   $Ø3
BAUD_48ØØ        equ   $Ø4
BAUD_96ØØ        equ   $Ø5
BAUD_192ØØ       equ   $Ø6
BAUD_MIDI        equ   $Ø7

FANFOLD          equ   $ØØ
SINGLE           equ   $8Ø
PICA             equ   $ØØØ
ELITE            equ   $4ØØ

FINE             equ   $8ØØ
DRAFT            equ   $ØØØ
LETTER           equ   $1ØØ

SIX_LPI          equ   $ØØØ
```

```
EIGHT_LPI        equ   $200

IMAGE_POSITIVE   equ   0
IMAGE_NEGATIVE   equ   1

ASPECT_HORIZ     equ   0
ASPECT_VERT      equ   1

SHADE_BW         equ   $00
SHADE_GREYSCALE  equ   $01
SHADE_COLOR      equ   $02

US_LETTER        equ   $00
US_LEGAL         equ   $10
N_TRACTOR        equ   $20
W_TRACTOR        equ   $30
CUSTOM           equ   $40
CUSTOM_NAME      equ   $00

ALPHA_P_101      equ   $01
BROTHER_15XL     equ   $02
CBM_MPS1000      equ   $03
DIAB_630         equ   $04
DIAB_ADV_D25     equ   $05
DIAB_C_150       equ   $06
EPSON            equ   $07
EPSON_JX_80      equ   $08
OKIMATE_20       equ   $09
QUME_LP_20       equ   $0A
HP_LASERJET      equ   $0B
HP_LASERJET_PLUS equ   $0C

;-------------------------------------------
;               Remember
;-------------------------------------------

00             rm_NextRemember ds.l    1      ;^Nächster Knoten
04             rm_RememberSize ds.l    1      ;Größe
08             rm_Memory       ds.l    1      ;^Adresse
0C             rm_SIZEOF       ds.w    0

;-------------------------------------------
;               Alerts
;-------------------------------------------

ALERT_TYPE      equ   $80000000
RECOVERY_ALERT  equ   $00000000
DEADEND_ALERT   equ   $80000000
```

A 4.4 Graphics

```
;zu Testzwecken importiert vom:
;==============================
MP_SIZE equ     $22      ;ports
LH_SIZE equ     $ØE      ;lists
MN_SIZE equ     $14      ;ports
IS_SIZE equ     $16      ;libraries
LIB_SIZE equ    $22
;===============================

;Layer-Structure
;---------------

ØØ   lr_Front            ds.l    1    ;^Layer ueber diesem
Ø4   lr_Back             ds.l    1    ;^Layer unter diesem
Ø8   lr_ClipRect         ds.l    1    ;^Clipping-Rechteck-Struktur
ØC   lr_RastPort         ds.l    1    ;^Rastport
1Ø   lr_MinX             ds.w    1    ;Clipping-Rechteck:
12   lr_MinY             ds.w    1
14   lr_MaxX             ds.w    1
16   lr_MaxY             ds.w    1
18   lr_Lock             ds.b    1    ;Task-Lock des Layers
19   lr_LockCount        ds.b    1    ;Anzahl Tasks zu
1A   lr_LayerLockCount   ds.b    1    ;  diesem Layer
1B   lr_reserved         ds.b    1
1C   lr_reserved1        ds.w    1
1E   lr_Flags            ds.w    1    ;16 Bit = Typ
2Ø   lr_SuperBitMap      ds.l    1    ;^Super-Bitmap
24   lr_SuperClipRect    ds.l    1    ;^ClipRect wenn S-Bitmap-Lay.
28   lr_Window           ds.l    1    ;^Intuition-Window
2C   lr_Scroll_ X        ds.w    1    ;Scro-Weite in Pixels
2E   lr_Scroll_Y         ds.w    1
3Ø   lr_LockPort         ds.b    MP_SIZE ;Name Msg-Port
52   lr_LockMessage      ds.b    MN_SIZE ;Msg-Struktur
66   lr_ReplyPort        ds.b    MP_SIZE ;Name Msg-Port
88   lr_1_LockMessage    ds.b    MN_SIZE ;Msg-Struktur
9C   lr_DamageList       ds.l    1    ;^Region-Struktur
AØ   lr__cliprects       ds.l    1    ;^Clip-Rect
A4   lr_LayerInfo        ds.l    1    ;^Layerinfo-Stuktur
A8   lr_LayerLocker      ds.l    1    ;^Task-Struktur
AC   lr_SuperSaverClipRects  ds.l    1 ;System-Use:
BØ   lr_cr              ds.l    1
B4   lr_cr2             ds.l    1
B8   lr_crnew           ds.l    1
BC   lr__p1             ds.l    1
CØ   lr_SIZEOF          ds.w    Ø

;Clip-Rect
;---------
ØØ   cr_Next           ds.l    1     ;^Nachfolger
Ø4   cr_Prev           ds.l    1     ;^Vorgaenger
```

```
Ø8    cr_LObs        ds.l    1      ;System-Use
ØC    cr_BitMap      ds.l    1      ;^Super-Bitmap
1Ø    cr_MinX        ds.w    1      ;Rechteck:
12    cr_MinY        ds.w    1
14    cr_MaxX        ds.w    1
16    cr_MaxY        ds.w    1
18    cr__p1         ds.l    1      ;System-Use:
1C    cr__p2         ds.l    1
2Ø    cr_reserved    ds.l    1
24    cr_Flags       ds.l    1
28    cr_SIZEOF      ds.w    Ø


ISLESSX equ 1
ISLESSY equ 2
ISGRTRX equ 4
ISGRTRY equ 8

;copper
;------

COPPER_MOVE      equ     Ø          ;Pseudo-Op-Codes
COPPER_WAIT      equ     1
CPRNXTBUF        equ     2
CPR_NT_LOF       equ     $8ØØØ
CPR_NT_SHT       equ     $4ØØØ

ØØ    ci_OpCode      ds.w    1      ;Op-Codes
Ø2    ci_nxtlist     ds.b    Ø
Ø2    ci_VWaitPos    ds.b    Ø
Ø2    ci_DestAddr    ds.b    2
Ø4    ci_HWaitPos    ds.b    Ø
Ø4    ci_DestData    ds.b    2
Ø6    ci_SIZEOF      ds.w    Ø

ØØ    crl_Next       ds.l    1      ;^aktuelle Copper-Liste
Ø4    crl_start      ds.l    1
Ø8    crl_MaxCount   ds.w    1
ØA    crl_SIZEOF     ds.w    Ø

;Copper-Liste
;------------
ØØ    cl_Next        ds.l    1      ;^Nachfolger
Ø4    cl__CopList    ds.l    1      ;System-Use
Ø8    cl_ViewPort    ds.l    1      ;
ØC    cl_CopIns      ds.l    1      ;
1Ø    cl_CopPtr      ds.l    1
14    cl_CopLStart   ds.l    1
18    cl_CopSStart   ds.l    1
1C    cl_Count       ds.w    1
1E    cl_MaxCount    ds.w    1
2Ø    cl_DyOffset    ds.w    1
```

```
22   cl_SIZEOF         ds.w    Ø

ØØ   ucl_Next          ds.l    1     ;Cop-List-Header
Ø4   ucl_FirstCopList  ds.l    1
Ø8   ucl_CopList       ds.l    1
ØC   ucl_SIZEOF        ds.w    Ø

ØØ   copinit_diagstrt   ds.b    8  ;interne Cop-Struktur
Ø8   copinit_sprstrtup  ds.b    8Ø
58   copinit_sprstop    ds.b    4
5C   copinit_SIZEOF     ds.w    Ø

;Gels (Grafik-Elemente)
;----------------------

SUSERFLAGS       equu    $ØF
VSB_VSPRITE      equ     Ø
VSF_VSPRITE      equ     1
VSB_SAVEBACK     equ     1
VSF_SAVEBACK     equ     2
VSB_OVERLAY      equ     2
VSF_OVERLAY      equ     4
VSB_MUSTDRAW     equ     3
VSF_MUSTDRAW     equ     8
VSB_BACKSAVED    equ     8
VSF_BACKSAVED    equ     $1ØØ
VSB_BOBUPDATE    equ     9
VSF_BOBUPDATE    equ     $2ØØ
VSB_GELGONE      equ     1Ø
VSF_GELGONE      equ     $4ØØ
VSB_VSOVERFLOW   equ     11
VSF_VSOVERFLOW   equ     $8ØØ

BUSERFLAGS       equ     $ØFF
BB_SAVEBOB       equ     Ø
BF_SAVEBOB       equ     1
BB_BOBISCOMP     equ     1
BF_BOBISCOMP     equ     2
BB_BWAITING      equ     8
BF_BWAITING      equ     $1ØØ
BB_BDRAWN        equ     9
BF_BDRAWN        equ     $2ØØ
BB_BOBSAWAY      equ     1Ø
BF_BOBSAWAY      equ     $4ØØ
BB_BOBNIX        equ     11
BF_BOBNIX        equ     $8ØØ
BB_SAVEPRESERVE  equ     12
BF_SAVEPRESERVE  equ     $1ØØØ
BB_OUTSTEP       equ     13
BF_OUTSTEP       equ     $2ØØØ

ANFRACSIZE       equ      6
```

```
ANIMHALF        equ     $2Ø
RINGTRIGGER     equ     1

;V-Sprite-Struktur (auch fuer Bobs)
;-----------------------------------
ØØ   vs_NextVSprite  ds.l   1      ;^Nachfolger
Ø4   vs_PrevVSprite  ds.l   1      ;^Vorgaenger
Ø8   vs_DrawPath     ds.l   1      ;System-Use
ØC   vs_ClearPath    ds.l   1      ;System-Use
1Ø   vs_Oldy         ds.w   1      ;Vorherige Pos. Y
12   vs_Oldx         ds.w   1      ;               X
14   vs_VSFlags      ds.w   1      ;Ist:
16   vs_Y            ds.w   1
18   vs_X            ds.w   1
1A   vs_Height       ds.w   1      ;Höhe des Sprite
1C   vs_Width        ds.w   1      ;Breite
1E   vs_Depth        ds.w   1      ;Bit-Planes
2Ø   vs_MeMask       ds.w   1      ;Masken fuer
22   vs_HitMask      ds.w   1      ;Kollisions-Handling
24   vs_ImageData    ds.l   1      ;^Daten
28   vs_BorderLine   ds.l   1      ;^Puffer
2C   vs_CollMask     ds.l   1      ;^Kollisions-Maske
3Ø   vs_SprColors    ds.l   1      ;^Color-Tabelle
34   vs_VSBob        ds.l   1      ;^Bob wenn Bob
38   vs_PlanePick    ds.b   1      ;Plane-Maske wenn Bob
39   vs_PlaneOnOff   ds.b   1      ;
3A   vs_SUserExt     ds.w   Ø      ;evtl. User-Extensions
3A   vs_SIZEOF       ds.w   Ø

;Bobs
;----
ØØ   bob_BobFlags    ds.w   1      ;Aspekt-Bits
Ø2   bob_SaveBuffer  ds.l   1      ;^Puffer
Ø6   bob_ImageShadow ds.l   1      ;^Shadowmask
ØA   bob_Before      ds.l   1      ;^Vorgaenger
ØE   bob_After       ds.l   1      ;^Nachfolger
12   bob_BobVSprite  ds.l   1      ;
16   bob_BobComp     ds.l   1
1A   bob_DBuffer     ds.l   1
1E   bob_BUserExt    ds.w   Ø
1E   bob_SIZEOF      ds.w   Ø

;Animations-Ablauf
;-----------------
ØØ   ac_CompFlags    ds.w   1      ;Typ-Bits
Ø2   ac_Timer        ds.w   1      ;Ist-Zeit
Ø4   ac_TimeSet      ds.w   1      ;Vorgabe
Ø6   ac_NextComp     ds.l   1      ;^Nachfolger
ØA   ac_PrevComp     ds.l   1      ;^Vorgaenger
ØE   ac_NextSeq      ds.l   1      ;dto in
12   ac_PrevSeq      ds.l   1      ;Zeichensequenz
16   ac_AnimCRoutine ds.l   1      ;^Exit-Routine (Ø)
```

```
1A    ac_YTrans       ds.w    1       ;Anfangsdistanz
1C    ac_XTrans       ds.w    1
1E    ac_HeadOb       ds.l    1       ;^AminOb-Structure
22    ac_AnimBob      ds.l    1       ;^Bob-Structure
26    ac_SIZE         ds.w    Ø

;Animations-Objekt
;-----------------
ØØ    ao_NextOb       ds.l    1       ;^Nachfolger
Ø4    ao_PrevOb       ds.l    1       ;^Vorgaenger
Ø8    ao_Clock        ds.l    1       ;Aufrufe
ØC    ao_AnOldY       ds.w    1       ;Alte Lage
ØE    ao_AnOldX       ds.w    1
1Ø    ao_AnY          ds.w    1       ;Ist
12    ao_AnX          ds.w    1
14    ao_YVel         ds.w    1       ;Speed
16    ao_XVel         ds.w    1
18    ao_XAccel       ds.w    1       ;Beschleunigung
1A    ao_YAccel       ds.w    1
1C    ao_RingYTrans   ds.w    1       ;Inkrements
1E    ao_RingXTrans   ds.w    1
2Ø    ao_AnimORoutine ds.l    1       ;^Routine
24    ao_HeadComp     ds.l    1       ;^erstes Objekt
28    ao_AUserExt     ds.w    Ø
28    ao_SIZEOF       ds.w    Ø

ØØ    dbp_BufY        ds.w    1       ;Zwischenpuffer
Ø2    dbp_BufX        ds.w    1
Ø4    dbp_BufPath     ds.l    1
Ø8    dbp_BufBuffer   ds.l    1
ØC    dbp_BufPlanes   ds.l    1
1Ø    dbp_SIZEOF      ds.w    Ø

;gfxbase
;-------

22    gb_ActiView        ds.l    1
26    gb_copinit         ds.l    1
2A    gb_cia             ds.l    1
2E    gb_blitter         ds.l    1
32    gb_LOFlist         ds.l    1
36    gb_SHFlist         ds.l    1
3A    gb_blthd           ds.l    1
3E    gb_blttl           ds.l    1
42    gb_bsblthd         ds.l    1
46    gb_bsblttl         ds.l    1
4A    gb_vbsrv           ds.b    IS_SIZE
6Ø    gb_timsrv          ds.b    IS_SIZE
76    gb_bltsrv          ds.b    IS_SIZE
8C    gb_TextFonts       ds.b    LH_SIZE
9A    gb_DefaultFont     ds.l    1
9E    gb_Modes           ds.w    1
```

```
A0   gb_VBlank          ds.b   1
A1   gb_Debug           ds.b   1
A2   gb_BeamSync        ds.w   1
A4   gb_system_bplcon0  ds.w   1
A6   gb_SpriteReserved  ds.b   1
A7   gb_bytereserved    ds.b   1
A8   gb_Flags           ds.w   1
AA   gb_BlitLock        ds.w   1
AC   gb_BlitNest        ds.w   1
AE   gb_BlitWaitQ       ds.b   LH_SIZE
BC   gb_BlitOwner       ds.l   1
C0   gb_TOF_WaitQ       ds.b   LH_SIZE
CE   gb_DisplayFlags    ds.w   1
D0   gb_SimpleSprites   ds.l   1
D4   gb_MaxDisplayRow   ds.w   1
D6   gb_reserved        ds.b   8
DE   gb_SIZE            ds.w   0


OWNBLITTERn     equ   0
QBOWNERn        equ   1
QBOWNER         equ   2

    ;GFX
;------------

BITSET  equ $8000
BITCLR  equ 0
AGNUS   equ 1
DENISE  equ 1

00   bm_BytesPerRow  ds.w   1
02   bm_Rows         ds.w   1
04   bm_Flags        ds.b   1
05   bm_Depth        ds.b   1
06   bm_Pad          ds.w   1
08   bm_Planes       ds.b   32
28   bm_SIZEOF       ds.w   0

00   ra_MinX         ds.w   1
02   ra_MinY         ds.w   1
04   ra_MaxX         ds.w   1
06   ra_MaxY         ds.w   1
08   ra_SIZEOF       ds.w   0

;layers
;------

00   lie_env            ds.b   52
34   lie_mem            ds.b   LH_SIZE
42   lie_FreeClipRects  ds.l   1
46   lie_blitbuff       ds.l   1
```

```
4A   lie_SIZEOF          ds.w    Ø

LMN_REGION     equ  -1

;memory
;------
memnode_succ     ds.l    1
memnode_pred     ds.l    1
memnode_where    ds.l    1
memnode_how_big  ds.l    1
memnode_SIZEOF   ds.w    Ø

;LayerInfo-Struktur
;-----------------
ØØ   li_top_layer        ds.l    1          ;^Layer oben
Ø4   li_check_lp         ds.l    1          ;System-Use:
Ø8   li_obs              ds.l    1
ØC   li_RP_ReplyPort     ds.b    MP_SIZE
2E   li_LockPort         ds.b    MP_SIZE
5Ø   li_Lock             ds.b    1
51   li_broadcast        ds.b    1
52   li_locknest         ds.b    1
53   li_pad              ds.b    1
54   li_Locker           ds.l    1
58   li_bytereserved     ds.b    2
5A   li_wordreserved     ds.b    4
5E   li_longreserved     ds.b    4
62   li_LayerInfo_extra  ds.l    1
66   li_SIZEOF           ds.w    Ø

NEWLAYERINFO_CALLED      equ 1

;rastport
;--------

ØØ   tr_RasPtr        ds.l    1
Ø4   tr_Size          ds.l    1
Ø8   tr_SIZEOF        ds.w    Ø

ØØ   gi_sprRsrvd       ds.b    1
Ø1   gi_Flags          ds.b    1
Ø2   gi_gelHead        ds.l    1
Ø6   gi_gelTail        ds.l    1
ØA   gi_nextLine       ds.l    1
ØE   gi_lastColor      ds.l    1
12   gi_collHandler    ds.l    1
16   gi_leftmost       ds.w    1
18   gi_rightmost      ds.w    1
1A   gi_topmost        ds.w    1
1C   gi_bottommost     ds.w    1
1E   gi_firstBlissObj  ds.l    1
22   gi_lastBlissObj   ds.l    1
```

```
26   gi_SIZEOF        ds.w    Ø

RPB_FRST_DOT     equ      Ø
RPF_FRST_DOT     equ      1
RPB_ONE_DOT      equ      1
RPF_ONE_DOT      equ      2
RPB_DBUFFER      equ      2
RPF_DBUFFER      equ      4
RPB_AREAOUTLINE equ      3
RPF_AREAOUTLINE equ      8
RPB_NOCROSSFILL equ      5
RPF_NOCROSSFILL equ      32

RP_JAM1          equ Ø
RP_JAM2          equ 1
RP_COMPLEMENT    equ 2
RP_INVERSVID     equ 4

RPB_TXSCALE      equ      Ø
RPF_TXSCALE      equ      1

;RastPort-Struktur
;-----------------
ØØ   rp_Layer         ds.l    1     ;^Layer
Ø4   rp_BitMap        ds.l    1     :^Bitmap
Ø8   rp_AreaPtrn      ds.l    1     ;^Fuellmuster
ØC   rp_TmpRas        ds.l    1     ;^Zwischenpuffer
1Ø   rp_AreaInfo      ds.l    1     ;^Info-Struktur
14   rp_GelsInfo      ds.l    1     ;^GelInfo-Struktur
18   rp_Mask          ds.b    1     ;Schreibmaske
19   rp_FgPen         ds.b    1     ;Vordergrund-Pen
1A   rp_BgPen         ds.b    1     ;Hintergrund-Pen
1B   rp_AOLPen        ds.b    1     ;Flood-Pen
1C   rp_DrawMode      ds.b    1     ;Zeichenmodus
1D   rp_AreaPtSz      ds.b    1     ;Worte Flood-Muster
1E   rp_Dummy         ds.b    1     ;Dummy
1F   rp_linpatcnt     ds.b    1     ;Poly-Count
2Ø   rp_Flags         ds.w    1     ;System-Use
22   rp_LinePtrn      ds.w    1     ;Linienmuster
24   rp_cp_x          ds.w    1     ;Pen-Position
26   rp_cp_y          ds.w    1
28   rp_minterms      ds.b    8     ;Blitter-Control
3Ø   rp_PenWidth      ds.w    1     ;Groesse Pen
32   rp_PenHeight     ds.w    1
34   rp_Font          ds.l    1     ;^Font
38   rp_AlgoStyle     ds.b    1     ;Text-Parms:
39   rp_TxFlags       ds.b    1
3A   rp_TxHeight      ds.w    1
3C   rp_TxWidth       ds.w    1
3E   rp_TxBaseline    ds.w    1
4Ø   rp_TxSpacing     ds.w    1
42   rp_RP_User       ds.l    1     ;^Reply-Port
```

```
46    rp_wordreserved ds.b    14
54    rp_longreserved ds.b    8
5C    rp_reserved     ds.b    8
64    rp_SIZEOF       ds.w    Ø

ØØ    ai_VctrTbl      ds.l    1
Ø4    ai_VctrPtr      ds.l    1
Ø8    ai_FlagTbl      ds.l    1
ØC    ai_FlagPtr      ds.l    1
1Ø    ai_Count        ds.w    1
12    ai_MaxCount     ds.w    1
14    ai_FirstX       ds.w    1
16    ai_FirstY       ds.w    1
18    ai_SIZEOF       ds.w    Ø

ONE_DOTn        equu    1
ONE_DOT         equ     $2
FRST_DOTn       equ     Ø
FRST_DOT        equ     1

;REGIONS
;-------

ØØ    rg_bounds           ds.b    ra_SIZEOF
Ø8    rg_RegionRectangle  ds.l    1
ØC    rg_SIZEOF           ds.w    Ø

ØØ    rr_Next         ds.l    1
Ø4    rr_Prev         ds.l    1
Ø8    rr_bounds       ds.b    ra_SIZEOF
1Ø    rr_SIZEOF       ds.w    Ø

;Sprites
;-------

ØØ    ss_posctldata   ds.l    1       ;^Daten Sprite
Ø4    ss_height       ds.w    1       ;Höhe
Ø6    ss_x            ds.w    1       ;aktuelle Position X
Ø8    ss_y            ds.w    1       ;                  Y
ØA    ss_num          ds.w    1       ;Sprite-Nummer (Ø..7)
ØC    ss_SIZEOF       ds.w    Ø

;Text
;----

FS_NORMAL       equ     Ø
FSB_EXTENDED    equ     3
FSF_EXTENDED    equ     8
FSB_ITALIC      equ     2
FSF_ITALIC      equ     4
FSB_BOLD        equ     1
FSF_BOLD        equ     2
```

```
FSB_UNDERLINED  equ     Ø
FSF_UNDERLINED  equ     1

FPB_ROMFONT      equ     Ø
FPF_ROMFONT      equ     1
FPB_DISKFONT     equ     1
FPF_DISKFONT     equ     2
FPB_REVPATH      equ     2
FPF_REVPATH      equ     4
FPB_TALLDOT      equ     3
FPF_TALLDOT      equ     8
FPB_WIDEDOT      equ     4
FPF_WIDEDOT      equ     16
FPB_PROPORTIONAL equ     5
FPF_PROPORTIONAL equ     32
FPB_DESIGNED     equ     6
FPF_DESIGNED     equ     64
FPB_REMOVED      equ     7
FPF_REMOVED      equ     128

ØØ    ta_Name         ds.l    1
Ø4    ta_YSize        ds.w    1
Ø6    ta_Style        ds.b    1
Ø7    ta_Flags        ds.b    1
Ø8    ta_SIZEOF       ds.w    Ø

14    tf_YSize        ds.w    1
16    tf_Style        ds.b    1
17    tf_Flags        ds.b    1
18    tf_XSize        ds.w    1
1A    tf_Baseline     ds.w    1
1C    tf_BoldSmear    ds.w    1
1E    tf_Accessors    ds.w    1
2Ø    tf_LoChar       ds.b    1
21    tf_HiChar       ds.b    1
22    tf_CharData     ds.l    1
26    tf_Modulo       ds.w    1
28    tf_CharLoc      ds.l    1
2C    tf_CharSpace    ds.l    1
3Ø    tf_CharKern     ds.l    1
34    tf_SIZEOF       ds.w    Ø

;View
;----

V_PFBA          equ     $4Ø
V_DUALPF        equ     $4ØØ
V_HIRES         equ     $8ØØØ
V_LAC           equ     4
V_HAM           equ     $8ØØ
V_SPRITES       equ     $4ØØØ
```

```
GENLOCK_VIDEO    equ 2

cm_Flags         ds.b    1
cm_Type          ds.b    1
cm_Count         ds.w    1
cm_ColorTable    ds.l    1
cm_SIZEOF        ds.w    0

;ViewPort-Struktur
;-----------------
00    vp_Next          ds.l    1       ;^Nachfolger
04    vp_ColorMap      ds.l    1       ;^
08    vp_DspIns        ds.l    1       ;
0C    vp_SprIns        ds.l    1
10    vp_ClrIns        ds.l    1
14    vp_UCopIns       ds.l    1
18    vp_DWidth        ds.w    1       ;Breite
1A    vp_DHeight       ds.w    1       ;Hoehe
1C    vp_DxOffset      ds.w    1
1E    vp_DyOffset      ds.w    1
20    vp_Modes         ds.w    1
22    vp_reserved      ds.w    1
24    vp_RasInfo       ds.l    1
28    vp_SIZEOF        ds.w    0

00    v_ViewPort       ds.l    1
04    v_LOFCprList     ds.l    1
08    v_SHFCprList     ds.l    1
0C    v_DyOffset       ds.w    1
0E    v_DxOffset       ds.w    1
10    v_Modes          ds.w    1
12    v_SIZEOF         ds.w    0

00    cp_collPtrs      ds.l    1
04    cp_SIZEOF        ds.w    0

00    ri_Next          ds.l    1
04    ri_BitMap        ds.l    1
08    ri_RxOffset      ds.w    1
0A    ri_RyOffset      ds.w    1
0C    ri_SIZEOF        ds.w    0
```

A 4.5 Devices

```
;importiert von exec:
;--------------------------
CMD_NONSTD       equ     9
IO_SIZE          equ     $20
IOSTD_SIZE       equ     $30
LN_SIZE          equ     $0E
MN_SIZE          equ     $14
```

```
TV_SIZE         equ     8
LIB_SIZE        equ     $22
MP_SIZE         equ     $22
pf_SIZEOF       equ     $E8
TC_SIZE         equ     $5C
LN_PRI          equ     9
;---------------------------

;Audio
;-----

ADHARD_CHANNELS equ   4

ADALLOC_MINPREC equ   -128
ADALLOC_MAXPREC equ   127

CMD_NONSTD      equ   9

ADCMD_FREE      equ   9
ADCMD_SETPREC   equ   10
ADCMD_FINISH    equ   11
ADCMD_PERVOL    equ   12
ADCMD_LOCK      equ   13
ADCMD_WAITCYCLE equ   14

ADCMDB_NOUNIT   equ   5
ADCMDF_NOUNIT   equ   32
ADCMD_ALLOCATE  equ   ADCMDF_NOUNIT

ADIOB_PERVOL            equ   4
ADIOF_PERVOL            equ   16
ADIOB_SYNCCYCLE         equ   5
ADIOF_SYNCCYCLE         equ   32
ADIOB_NOWAIT            equ   6
ADIOF_NOWAIT            equ   64
ADIOB_WRITEMESSAGE      equ   7
ADIOF_WRITEMESSAGE      equ   128

ADIOERR_NOALLOCATION    equ   -10
ADIOERR_ALLOCFAILED     equ   -11
ADIOERR_CHANNELSTOLEN   equ   -12

20              ioa_AllocKey    ds.w    1
22              ioa_Data        ds.l    1
26              ioa_Length      ds.l    1
2A              ioa_Period      ds.w    1
2C              ioa_Volume      ds.w    1
2E              ioa_Cycles      ds.w    1
30              ioa_WriteMsg    ds.b    MN_SIZE
                ioa_SIZEOF      equ     $44
```

```
;bootblock
;---------

ØØ            BB_ID           ds.b    4
Ø4            BB_CHKSUM       ds.l    1
Ø8            BB_DOSBLOCK     ds.l    1
              BB_ENTRY        equ     $ØC
              BB_SIZE         equ     $ØC

BOOTSECTS      equ   2

BBNAME_DOS     equ   444F53ØØ  ; 'DOS'<<8
BBNAME_KICK    equ   4B49434B  ; 'KICK'

;CLIPBOARD
;---------

CBERR_OBSOLETEID        equ    1

ØØ            cu_Node         ds.b    LN_SIZE
ØE            cu_UnitNum      ds.l    1

ØØ            io_Message      ds.b    MN_SIZE
14            io_Device       ds.l    1
18            io_Unit         ds.l    1
1C            io_Command      ds.w    1
1E            io_Flags        ds.b    1
1F            io_Error        ds.b    1
2Ø            io_Actual       ds.l    1
24            io_Length       ds.l    1
28            io_Data         ds.l    1
2C            io_Offset       ds.l    1
3Ø            io_ClipID       ds.l    1
              iocr_SIZEOF     equ     $34

PRIMARY_CLIP   equ   Ø

ØØ            sm_Msg             ds.b    MN_SIZE
14            sm_Unit            ds.w    1
16            sm_ClipID          ds.l    1
              satisfyMsg_SIZEOF equ     $1A

;CONSOLE
;-------

CD_ASKKEYMAP   equ   9
CD_SETKEYMAP   equ   1Ø

SGR_PRIMARY    equ   Ø
SGR_BOLD       equ   1
SGR_ITALIC     equ   3
SGR_UNDERSCORE equ   4
```

```
SGR_NEGATIVE    equ   7
SGR_BLACK       equ   30
SGR_RED         equ   31
SGR_GREEN       equ   32
SGR_YELLOW      equ   33
SGR_BLUE        equ   34
SGR_MAGENTA     equ   35
SGR_CYAN        equ   36
SGR_WHITE       equ   37
SGR_DEFAULT     equ   39
SGR_BLACKBG     equ   40
SGR_REDBG       equ   41
SGR_GREENBG     equ   42
SGR_YELLOWBG    equ   43
SGR_BLUEBG      equ   44
SGR_MAGENTABG   equ   45
SGR_CYANBG      equ   46
SGR_WHITEBG     equ   47
SGR_DEFAULTBG   equ   49
SGR_CLR0        equ   30
SGR_CLR1        equ   31
SGR_CLR2        equ   32
SGR_CLR3        equ   33
SGR_CLR4        equ   34
SGR_CLR5        equ   35
SGR_CLR6        equ   36
SGR_CLR7        equ   37
SGR_CLR0BG      equ   40
SGR_CLR1BG      equ   41
SGR_CLR2BG      equ   42
SGR_CLR3BG      equ   43
SGR_CLR4BG      equ   44
SGR_CLR5BG      equ   45
SGR_CLR6BG      equ   46
SGR_CLR7BG      equ   47

DSR_CPR         equ   6

CTC_HSETTAB     equ   0
CTC_HCLRTAB     equ   2
CTC_HCLRTABSALL equ   5

TBC_HCLRTAB     equ   0
TBC_HCLRTABSALL equ   3

;gameport
;--------

GPD_READEVENT     equ     9
GPD_ASKCTYPE      equ     10
GPD_SETCTYPE      equ     11
GPD_ASKTRIGGER    equ     12
```

```
GPD_SETTRIGGER     equ     13
GPTB_DOWNKEYS      equ     Ø
GPTF_DOWNKEYS      equ     1
GPTB_UPKEYS        equ     1
GPTF_UPKEYS        equ     2

ØØ             gpt_Keys         ds.w     1
Ø2             gpt_Timeout      ds.w     1
Ø4             gpt_XDelta       ds.w     1
Ø6             gpt_YDelta       ds.w     1
               gpt_SIZEOF       equ      8

GPCT_ALLOCATED           equ    -1
GPCT_NOCONTROLLER        equ    Ø
GPCT_MOUSE               equ    1
GPCT_RELJOYSTICK         equ    2
GPCT_ABSJOYSTICK         equ    3
GPDERR_SETCTYPE          equ    1

;input
;-----

IND_ADDHANDLER       equ     9
IND_REMHANDLER       equ     1Ø
IND_WRITEEVENT       equ     11
IND_SETTHRESH        equ     12
IND_SETPERIOD        equ     13
IND_SETMPORT         equ     14
IND_SETMTYPE         equ     15
IND_SETMTRIG         equ     16

;INPUTEVENT
;----------

IECLASS_NULL             equ    Ø
IECLASS_RAWKEY           equ    1
IECLASS_RAWMOUSE         equ    2
IECLASS_EVENT            equ    3
IECLASS_POINTERPOS       equ    4
IECLASS_TIMER            equ    6
IECLASS_GADGETDOWN       equ    7
IECLASS_GADGETUP         equ    8
IECLASS_RequESTER        equ    9
IECLASS_MENULIST         equ    1Ø
IECLASS_CLOSEWINDOW      equ    11
IECLASS_SIZEWINDOW       equ    12
IECLASS_REFRESHWINDOW    equ    13
IECLASS_NEWPREFS         equ    14
IECLASS_DISKREMOVED      equ    15
IECLASS_DISKINSERTED     equ    16
IECLASS_ACTIVEWINDOW     equ    17
IECLASS_INACTIVEWINDOW   equ    18
```

```
IECLASS_MAX              equ   $12

IECODE_UP_PREFIX         equ   $80
IECODEB_UP_PREFIX        equ   7
IECODE_KEY_CD_FIRST      equ   0
IECODE_KEY_CD_LAST       equ   $77
IECODE_COMM_CD_FIRST     equ   $78
IECODE_COMM_CODE_LAST    equ   $7F

IECODE_C0_FIRST          equ   $00
IECODE_C0_LAST           equ   $1F
IECODE_ASCII_FIRST       equ   $20
IECODE_ASCII_LAST        equ   $7E
IECODE_ASCII_DEL         equ   $7F
IECODE_C1_FIRST          equ   $80
IECODE_C1_LAST           equ   $9F
IECODE_LATIN1_FIRST      equ   $A0
IECODE_LATIN1_LAST       equ   $FF
IECODE_LBUTTON           equ   $68
IECODE_RBUTTON           equ   $69
IECODE_MBUTTON           equ   $6A
IECODE_NOBUTTON          equ   $FF

IECODE_NEWACTIVE         equ   1
IECODE_REQSET            equ   1
IECODE_REQCLEAR          equ   0

IequALIFIER_LSHIFT           equ   1
IequALIFIERB_LSHIFT          equ   0
IequALIFIER_RSHIFT           equ   2
IequALIFIERB_RSHIFT          equ   1
IequALIFIER_CAPSLOCK         equ   4
IequALIFIERB_CAPSLOCK        equ   2
IequALIFIER_CONTROL          equ   8
IequALIFIERB_CONTROL         equ   3
IequALIFIER_LALT             equ   16
IequALIFIERB_LALT            equ   4
IequALIFIER_RALT             equ   32
IequALIFIERB_RALT            equ   5
IequALIFIER_LCOMMAND         equ   64
IequALIFIERB_LCOMMAND        equ   6
IequALIFIER_RCOMMAND         equ   128
IequALIFIERB_RCOMMAND        equ   7
IequALIFIER_NUMERICPAD       equ   $0100
IequALIFIERB_NUMERICPAD      equ   8
IequALIFIER_REPEAT           equ   $0200
IequALIFIERB_REPEAT          equ   9
IequALIFIER_INTERRUPT        equ   $0400
IequALIFIERB_INTERRUPT       equ   10
IequALIFIER_MULTIBROADCAST   equ   $0800
IequALIFIERB_MULTIBROADCAST equ   11
IequALIFIER_LBUTTON          equ   $1000
```

```
IequALIFIERB_LBUTTON        equ   12
IequALIFIER_RBUTTON         equ   $2ØØØ
IequALIFIERB_RBUTTON        equ   13
IequALIFIER_MBUTTON         equ   $4ØØØ
IequALIFIERB_MBUTTON        equ   14
IequALIFIER_RELATIVEMOUSE   equ   $8ØØØ
IequALIFIERB_RELATIVEMOUSE  equ   15

ØØ              ie_NextEvent     ds.l    1
Ø4              ie_Class         ds.b    1
Ø5              ie_SubClass      ds.b    1
Ø6              ie_Code          ds.w    1
Ø8              ie_Qualifier     ds.w    1
                ie_EventAddress  equ     $ØA
ØA              ie_X             ds.w    1
Øc              ie_Y             ds.w    1

ØE              ie_TimeStamp     ds.b    TV_SIZE
16              ie_SIZEOF        ds.w    Ø

;KEYBOARD
;--------

KBD_READEVENT          equ     9
KBD_READMATRIX         equ     1Ø
KBD_ADDRESETHANDLER    equ     11
KBD_REMRESETHANDLER    equ     12
KBD_RESETHANDLERDONE   equ     13

;Keymap
;------

ØØ              km_LoKeyMapTypes ds.l    1
Ø4              km_LoKeyMap      ds.l    1
Ø8              km_LoCapsable    ds.l    1
ØC              km_LoRepeatable  ds.l    1
1Ø              km_HiKeyMapTypes ds.l    1
14              km_HiKeyMap      ds.l    1
18              km_HiCapsable    ds.l    1
1C              km_HiRepeatable  ds.l    1
                km_SIZEOF        equ     $2Ø

KCB_NOP         equ   7
KCF_NOP         equ   $8Ø

KC_NOQUAL       equ   Ø
KC_VANILLA      equ   7
KCF_SHIFT       equ   1
KCF_ALT         equ   2
KCB_CONTROL     equ   2
KCF_CONTROL     equ   4
KCB_DOWNUP      equ   3
```

```
KCF_DOWNUP      equ   8

KCB_STRING      equ   6
KCB_STRING      equ   64

;Narrator
;========

DEFPITCH        equ       110
DEFRATE         equ       150
DEFVOL          equ       64
DEFFREQ         equ       22200
NATURALF0       equ       0
ROBOTICF0       equ       1
MALE            equ       0
FEMALE          equ       1
DEFSEX          equ       MALE
DEFMODE         equ       NATURALF0

MINRATE         equ       40
MAXRATE         equ       400
MINPITCH        equ       65
MAXPITCH        equ       320
MINFREQ         equ       5000
MAXFREQ         equ       28000
MINVOL          equ       0
MAXVOL          equ       64

ND_NotUsed      equ       -1
ND_NoMem        equ       -2
ND_NoAudLib     equ       -3
ND_MakeBad      equ       -4
ND_UnitErr      equ       -5
ND_CantAlloc    equ       -6
ND_Unimpl       equ       -7
ND_NoWrite      equ       -8
ND_Expunged     equ       -9
ND_PhonErr      equ       -20
ND_RateErr      equ       -21
ND_PitchErr     equ      -22
ND_SexErr       equ      -23
ND_ModeErr      equ      -24
ND_FreqErr      equ      -25
ND_VolErr       equ      -26

30             NDI_RATE        ds.w     1
32             NDI_PITCH       ds.w     1
34             NDI_MODE        ds.w     1
36             NDI_SEX         ds.w     1
38             NDI_CHMASKS     ds.l     1
3C             NDI_NUMMASKS    ds.w     1
3E             NDI_VOLUME      ds.w     1
```

```
40          NDI_SAMPFREQ    ds.w    1
42          NDI_MOUTHS      ds.b    1
43          NDI_CHANMASK    ds.b    1
44          NDI_NUMCHAN     ds.b    1
45          NDI_PAD         ds.b    1
            NDI_SIZE        equ     $46

46          MRB_WIDTH       ds.b    1
47          MRB_HEIGHT      ds.b    1
48          MRB_SHAPE       ds.b    1
49          MRB_PAD         ds.b    1
4A          MRB_SIZE        equ     $4A

;PARALLEL
;--------

ParErr_DevBusy          equ     1
ParErr_BufTooBig        equ     2
ParErr_InvParam         equ     3
ParErr_LineErr          equ     4
ParErr_NotOpen          equ     5
ParErr_PortReset        equ     6
ParErr_InitErr          equ     7

PDCMD_QUERY             equ     CMD_NONSTD
PDCMD_SETPARAMS         equ     CMD_NONSTD+1
Par_DEVFINISH           equ     10


PARB_SHARED     equ     5
PARF_SHARED     equ     32
PARB_RAD_BOOGIE equ     3
PARF_RAD_BOOGIE equ     8
PARB_EOFMODE    equ     1
PARF_EOFMODE    equ     2

IOPARB_QUEUED   equ     6
IOPARF_QUEUED   equ     128
IOPARB_ABORT    equ     5
IOPARF_ABORT    equ     32
IOPARB_ACTIVE   equ     4
IOPARF_ACTIVE   equ     16
IOPTB_RWDIR     equ     3
IOPTF_RWDIR     equ     8
IOPTB_PBUSY     equ     2
IOPTF_PBUSY     equ     4
IOPTB_PAPEROUT  equ     1
IOPTF_PAPEROUT  equ     2
IOPTB_PSEL      equ     0
IOPTF_PSEL      equ     1

00          PTERMARRAY_0    ds.l    1
```

```
Ø4              PTERMARRAY_1    ds.l    1
Ø8              PTERMARRAY_SIZE ds.w    Ø

3Ø              IO_PEXTFLAGS    ds.l    1
34              IO_PARSTATUS    ds.b    1
35              IO_PARFLAGS     ds.b    1
36              IO_PTERMARRAY   ds.b    PTERMARRAY_SIZE
3E              IOEXTPar_SIZE   equ     $3E

;Serial
;------

SER_CTL         equ     $1113ØØØØ
SER_DBAUD       equ     96ØØ


SDCMD_QUERY     equ     9
SDCMD_BREAK     equ     1Ø
SDCMD_SETPARAMS equ     CMD_NONSTD+2
SER_DEVFINISH   equ     11

SERB_XDISABLED  equ     7
SERF_XDISABLED  equ     128
SERB_EOFMODE    equ     6
SERF_EOFMODE    equ     64
SERB_SHARED     equ     5
SERF_SHARED     equ     32
SERB_RAD_BOOGIE equ     4
SERF_RAD_BOOGIE equ     16
SERB_QUEUEDBRK  equ     3
SERF_QUEUEDBRK  equ     8
SERB_7WIRE      equ     2
SERF_7WIRE      equ     4
SERB_PARTY_ODD  equ     1
SERF_PARTY_ODD  equ     2
SERB_PARTY_ON   equ     Ø
SERF_PARTY_ON   equ     1

IOSERB_QUEUED    equ     6
IOSERF_QUEUED    equ     64
IOSERB_ABORT     equ     5
IOSERF_ABORT     equ     32
IOSERB_ACTIVE    equ     4
IOSERF_ACTIVE    equ     16
IOSTB_XOFFREAD   equ     4
IOSTF_XOFFREAD   equ     16
IOSTB_XOFFWRITE  equ     3
IOSTF_XOFFWRITE  equ     8
IOSTB_READBREAK  equ     2
IOSTF_READBREAK  equ     4
IOSTB_WROTEBREAK equ     1
IOSTF_WROTEBREAK equ     2
```

```
IOSTB_OVERRUN    equ    Ø
IOSTF_OVERRUN    equ    1

ØØ              TERMARRAY_Ø      ds.l     1
Ø4              TERMARRAY_1      ds.l     1
                TERMARRAY_SIZE   equ      8

3Ø              IO_CTLCHAR       ds.l     1
34              IO_RBUFLEN       ds.l     1
38              IO_EXTFLAGS      ds.l     1
3C              IO_BAUD          ds.l     1
4Ø              IO_BRKTIME       ds.l     1
44              IO_TERMARRAY     ds.b     TERMARRAY_SIZE
4C              IO_READLEN       ds.b     1
4D              IO_WRITELEN      ds.b     1
4E              IO_STOPBITS      ds.b     1
4F              IO_SERFLAGS      ds.b     1
5Ø              IO_STATUS        ds.w     1
                IOEXTSER_SIZE    equ      $52

SerErr_DevBusy          equ     1
SerErr_BaudMismatch     equ     2
SerErr_InvBaud          equ     3
SerErr_BufErr           equ     4
SerErr_InvParam         equ     5
SerErr_LineErr          equ     6
SerErr_NotOpen          equ     7
SerErr_PortReset        equ     8
SerErr_ParityErr        equ     9
SerErr_InitErr          equ     1Ø
SerErr_TimerErr         equ     11
SerErr_BufOverflow      equ     12
SerErr_NoDSR            equ     13
SerErr_NoCTS            equ     14
SerErr_DetectedBreak    equ     15

;timer
;-----

UNIT_MICROHZ    equ   Ø
UNIT_VBLANK     equ   1

ØØ              TV_SECS          ds.l     1
Ø4              TV_MICRO         ds.l     1
                TV_SIZE          equ      8

2Ø              IOTV_TIME        ds.b     TV_SIZE
28              IOTV_SIZE        equ      $2Ø

TR_ADDRequEST     equ     9
TR_GETSYSTIME     equ     1Ø
TR_SETSYSTIME     equ     11
```

```
PRD_RAWWRITE      equ     9
PRD_PRTCOMMAND    equ     10
PRD_DUMPRPORT     equ     11

aRIS     equ    0
aRIN     equ    1
aIND     equ    2
aNEL     equ    3
aRI      equ    4

aSGR0    equ    5
aSGR3    equ    6
aSGR23   equ    7
aSGR4    equ    8
aSGR24   equ    9
aSGR1    equ    10
aSGR22   equ    11
aSFC     equ    12
aSBC     equ    13

aSHORP0  equ    14
aSHORP2  equ    15
aSHORP1  equ    16
aSHORP4  equ    17
aSHORP3  equ    18
aSHORP6  equ    19
aSHORP5  equ    20

aDEN6    equ    21
aDEN5    equ    22
aDEN4    equ    23
aDEN3    equ    24
aDEN2    equ    25
aDEN1    equ    26

aSUS2    equ    27
aSUS1    equ    28
aSUS4    equ    29
aSUS3    equ    30
aSUS0    equ    31
aPLU     equ    32
aPLD     equ    33

aFNT0    equ    34
aFNT1    equ    35
aFNT2    equ    36
aFNT3    equ    37
aFNT4    equ    38
aFNT5    equ    39
aFNT6    equ    40
aFNT7    equ    41
aFNT8    equ    42
```

```
aFNT9   equ   43
aFNT1Ø  equ   44

aPROP2  equ   45
aPROP1  equ   46
aPROPØ  equ   47
aTSS    equ   48

aJFY5   equ   49
aJFY7   equ   5Ø
aJFY6   equ   51
aJFYØ   equ   52
aJFY2   equ   53
aJFY3   equ   54

aVERPØ  equ   55
aVERP1  equ   56
aSLPP   equ   57
aPERF   equ   58
aPERFØ  equ   59

aLMS    equ   6Ø
aRMS    equ   61
aTMS    equ   62
aBMS    equ   63
aSTBM   equ   64
aSLRM   equ   65
aCAM    equ   66

aHTS    equ   67
aVTS    equ   68
aTBCØ   equ   69
aTBC3   equ   7Ø
aTBC1   equ   71
aTBC4   equ   72
aTBCALL equ   73
aTBSALL equ   74
aEXTEND equ   75

2Ø            io_PrtCommand   ds.w    1
22            io_ParmØ        ds.b    1
23            io_Parm1        ds.b    1
24            io_Parm2        ds.b    1
25            io_Parm3        ds.b    1
              iopcr_SIZEOF    equ     $26

2Ø            io_RastPort     ds.l    1
24            io_ColorMap     ds.l    1
28            io_Modes        ds.l    1
2C            io_SrcX         ds.w    1
2E            io_SrcY         ds.w    1
3Ø            io_SrcWidth     ds.w    1
```

```
32          io_SrcHeight    ds.w    1
34          io_DestCols     ds.l    1
38          io_DestRows     ds.l    1
3C          io_Special      ds.w    1
            iodrpr_SIZEOF   equ     $3E

SPECIAL_MILCOLS         equ   1
SPECIAL_MILROWS         equ   2
SPECIAL_FULLCOLS        equ   4
SPECIAL_FULLROWS        equ   8
SPECIAL_FRACCOLS        equ   16
SPECIAL_FRACROWS        equ   32

22          dd_Segment      ds.l    1
26          dd_ExecBase     ds.l    1
2A          dd_CmdVectors   ds.l    1
2E          dd_CmdBytes     ds.l    1
32          dd_NumCommands  ds.w    1
            dd_SIZEOF       equ     $34

du_Flags        equ   LN_PRI

IOB_QUEUED      equ     4
IOF_QUEUED      equ     16
IOB_CURRENT     equ     5
IOF_CURRENT     equ     32
IOB_SERVICING   equ     6
IOF_SERVICING   equ     64
IOB_DONE        equ     7
IOF_DONE        equ     128

DUB_STOPPED     equ     Ø
DUF_STOPPED     equ     1Ø

P_PRIORITY      equ     Ø
P_STKSIZE       equ     $8ØØ

PB_IORØ         equ     Ø
PF_IORØ         equ     1
PB_IOR1         equ     1
PF_IOR1         equ     2
PB_EXPUNGED     equ     7
PF_EXPUNGED     equ     128

34          pd_Unit             ds.b    MP_SIZE
56          pd_PrinterSegment   ds.l    1
5A          pd_PrinterType      ds.w    1
5C          pd_SegmentData      ds.l    1
6Ø          pd_PrintBuf         ds.l    1
64          pd_PWrite           ds.l    1
68          pd_PBothReady       ds.l    1
```

```
PPCB_GFX        equ     Ø
PPCF_GFX        equ     1
PPCB_COLOR      equ     1
PPCF_COLOR      equ     2

PPC_BWALPHA     equ   Ø
PPC_BWGFX       equ   1
PPC_COLORGFX    equ   3

PCC_BW          equ   1
PCC_YMC         equ   2
PCC_YMC_BW      equ   3
PCC_YMCB        equ   4

ØØ             ped_PrinterName  ds.l    1
Ø4             ped_Init         ds.l    1
Ø8             ped_Expunge      ds.l    1
ØC             ped_Open         ds.l    1
1Ø             ped_Close        ds.l    1
14             ped_PrinterClass ds.b    1
15             ped_ColorClass   ds.b    1
16             ped_MaxColumns   ds.b    1
17             ped_NumCharSets  ds.b    1
18             ped_NumRows      ds.w    1
1A             ped_MaxXDots     ds.l    1
1E             ped_MaxYDots     ds.l    1
22             ped_XDotsInch    ds.w    1
24             ped_YDotsInch    ds.w    1
26             ped_Commands     ds.l    1
2A             ped_DoSpecial    ds.l    1
2E             ped_Render       ds.l    1
32             ped_TimeoutSecs  ds.l    1
               ped_SIZEOF       equ     $36

ØØ             ps_NextSegment   ds.l    1
Ø4             ps_runAlert      ds.l    1
Ø8             ps_Version       ds.w    1
ØA             ps_Revision      ds.w    1
               ps_PED           equ    $ØC

SPECIAL_ASPECT          equ   $8Ø
SPECIAL_DENSITYMASK     equ   $FØØ
SPECIAL_DENSITY1        equ   $1ØØ
SPECIAL_DENSITY2        equ   $2ØØ
SPECIAL_DENSITY3        equ   $3ØØ
SPECIAL_DENSITY4        equ   $4ØØ

PDERR_CANCEL            equ   1
PDERR_NOTGRAPHICS       equ   2
PDERR_INVERTHAM         equ   3
PDERR_BADDIMENSION      equ   4
PDERR_DIMENSIONOVFLOW   equ  5
```

```
PDERR_INTERNALMEMORY     equ   6
PDERR_BUFFERMEMORY       equ   7

;TRACKDISK
;---------

NUMCYLS         equ   8Ø
MAXCYLS         equ   NUMCYLS+2Ø
NUMSECS         equ   11
NUMHEADS        equ   2
MAXRETRY        equ   1Ø
NUMTRACKS       equ   NUMCYLS*NUMHEADS
NUMUNITS        equ   4

TD_SECTOR       equ   512
TD_SECSHIFT     equ   9

TDB_EXTCOM      equ     15
TDF_EXTCOM      equ     $8ØØØ

;Commands
TD_MOTOR        equ     9
TD_SEEK         equ     1Ø
TD_FORMAT       equ     11
TD_REMOVE       equ     12
TD_CHANGENUM    equ     13
TD_CHANGESTATE  equ     14
TD_PROTSTATUS   equ     15
TD_LASTCOMM     equ     15

;Extended Commands (mit extendedt IORequest-Block)
ETD_WRITE       equ   3
ETD_READ        equ   2
ETD_MOTOR       equ   9
ETD_SEEK        equ   1Ø
ETD_FORMAT      equ   11
ETD_UPDATE      equ   4
ETD_CLEAR       equ   5

;IORequest-Block-Extension
3Ø             IOTD_COUNT      ds.l     1
34             IOTD_SECLABEL   ds.l     1
               IOTD_SIZE       equ      $38

TD_LABELSIZE             equ   16

;Error-Codes (in IOActual)
TDERR_NotSpecified       equ   2Ø
TDERR_NoSecHdr           equ   21
TDERR_BadSecPreamble     equ   22
TDERR_BadSecID           equ   23
TDERR_BadHdrSum          equ   24
```

```
TDERR_BadSecSum         equ   25
TDERR_TooFewSecs        equ   26
TDERR_BadSecHdr         equ   27
TDERR_WriteProt         equ   28
TDERR_DiskChanged       equ   29
TDERR_SeekError         equ   30
TDERR_NoMem             equ   31
TDERR_BadUnitNum        equ   32
TDERR_BadDriveType      equ   33
TDERR_DriveInUse        equ   34
```

Anhang A5: CLI

- Einführung in das CLI
- Einrichten der Arbeitsdisketten
- Include-Tips
- Tips und Tricks zum CLI

Dieser Anhang will Sie schnell soweit mit dem CLI bekanntmachen, wie es für die ersten Schritte in der Assembler-Programmierung erforderlich ist. Darüber hinaus möchte ich auf das Handbuch von Commodore und das Amiga-Buch von M. Breuer (Markt & Technik) verweisen. Der Abschnitt über Tips und Tricks setzt schon voraus, daß Sie die etwas komplizierteren CLI-Dinge anhand dieser Literatur verstanden haben.

CLI heißt Command Line Interpreter. Auf gut deutsch: Sie geben Kommandos über die Tastatur ein, das CLI interpretiert diese Kommandos und führt sie aus. Praktisch liegt hiermit die Bedienoberfläche der CP/M- oder MS-DOS-Computer vor, nur ist das CLI deutlich leistungsfähiger als zum Beispiel MS-DOS.

HFS

Jedes DOS kümmert sich in erster Linie um Files (Dateien). Wie diese Files auf der Disk angeordnet und aufzufinden sind, bestimmt das File-System. Der Amiga hat ein hierarchisches File-System, kurz HFS genannt.

Im HFS geht es um die Ablage von Files (Dateien), neuerdings auch Dokumente genannt. Der Vergleich mit einem Büro ist sehr zutreffend, wenn man vereinbart:
- Die Disk (Diskette oder Harddisk) ist das Büro
- Ein Directory ist ein Schrank
- Ein Unterdirectory ist eine Schublade in einem Schrank
- Ein Unter-Unterdirectory ist eine Schublade in einer Schublade
- Ein Dokument kann an beliebiger Stelle liegen:
  → mitten im Büro (auf dem Fußboden)
  → im Schrank (nicht in einer Schublade)
  → in einer Schublade
  → in einer Schublade, die in einer Schublade ist

Im Unterschied zum Schrankmodell kann man nahezu beliebig oft in eine Schublade immer wieder noch eine Schublade packen, in diese noch eine usw., wobei jedoch die

innere Schublade nicht kleiner sein muß, als die sie umgebende, genauer: Die Größe ist nicht definiert. Sie können in eine Schublade so lange Files packen, bis die Disk voll ist.

Namen von Disks

Sowohl die Disk als auch die Schubladen haben Namen, die Sie vergeben können. Ein Diskname endet immer mit einem Doppelpunkt. Sie können eine Disk aber nicht nur mit ihrem Namen ansprechen (manche hat gar keinen) sondern zusätzlich auch mit dem Gerätenamen. Dafür gilt:

DF0: = Diskette 0
DF1 = Diskette 1
JH0 = Harddisk 0 (Janus Harddisk)

Es gibt noch einen Namen, und das ist

SYS:

SYS: bezeichnet immer die Boot-Diskette. Hat Ihre Boot-Diskette zum Beispiel den Namen WBENCH, so können Sie die ansprechen mit

WBENCH:
oder DF0:
oder SYS:

Namen von Directories und Files

Eine Schublade wird fachmännisch Directory genannt. In den CLI-Befehlen taucht dafür auch das Kürzel DIR oder D auf.

Grundsätzlich gibt es keinen Unterschied, den Sie bei der Namensvergabe beachten müssen. Es dürfen sogar ein File und das Directory, in dem sich der File befindet, denselben Namen haben. Im allgemeinen erkennt man jedoch einen Directory-Namen am nachfolgenden Schrägstrich.

Pfadnamen

Letztendlich wirken die meisten Operationen immer auf einen File. Nehmen wir folgendes Beispiel:

Auf der Diskette im Drive DF0:
gibt es ein Directory namens ASSEMBLER
darin ein Directory namens SOURCES
und darin einen File namens TEST.S

Dann können Sie den File TEST.S ansprechen mit

```
DFØ:ASSEMBLER/SOURCES/TEST.S
```

Diese Gebilde nennt man Pfadnamen. Der sicherste Weg zu einem File ist immer der vollständige Pfadname. Da dies jedoch eine Menge Tipperei bedeutet, gibt es einige Methoden der Abkürzung, wie wir noch sehen werden.

CLI-Befehle

CLI-Befehle werden eingetippt und mit Return abgeschlossen. Der einfachste CLI-Befehl heißt DIR. DIR gibt das aktuelle Directory auf dem Schirm aus. Solange Sie keinen Pfadnamen angeben, wirken alle Befehle immer im aktuellen Directory. Die Anzeige des aktuellen Directory erreichen Sie mit CD.

CD (Change Directory) mit einem Pfadnamen ändert das aktuelle Directory. Tippen Sie (immer mit Return zum Schluß):

Befehl	Wirkung
CD df0:	Sie sind ganz oben (im Root-Directory)
DIR	Anzeige aller Files und DIRs auf dieser Ebene
MAKEDIR xyz	Anlegen eines neuen Directory, Name xyz
CD xyz	Sie sind im neuen Directory

Einrichten einer Arbeitsdiskette

Legen Sie zuerst eine Kopie der Workbench-Diskette an (besser zwei). Arbeiten Sie nur mit einer Kopie! Nach dem Start und dem Öffnen der Workbench-Disk sollten Sie ein Icon mit dem Namen CLI sehen. Wenn nicht, rufen Sie »Preferences« (durch Anklicken) auf und klicken Sie dort »CLI ON«; dann diesen Zustand sichern. Ab jetzt sollte bei jedem Start das CLI-Icon sichtbar sein.

Das CLI wird durch Anklicken gestartet. Es erscheint ein eigenes Window, das in der Regel zu klein ist. Ziehen Sie auf fast volle Schirmgröße. Bei dieser Gelegenheit ein Tip: Programmieren Sie niemals Fenster auf die volle Schirmgröße. Es könnte sein, daß durch Schwankungen der Monitore oder der Stromversorgung dann Teile des Fensters nicht mehr sichtbar sind.

Um gleich beim Start in das CLI zu gelangen, müssen Sie die Startup-Sequence ändern. Das ist ein File (reiner Text) mit CLI-Befehlen, die bei jedem Start automatisch ausgeführt werden. Wie alle diese Batch-Files (Ausführung sonst mit EXECUTE Filename) befindet er sich im s-Directory. Gehen Sie mit »CD s« in dieses Directory, laden

in den Editor (ED, siehe Handbuch) den File »startup-sequence«, entfernen die meistens letzte Zeile der Form »endcli >nil:« und speichern den File wieder ab.

Nun greifen Sie zur Panik-Taste (Control plus beide Amiga-As gleichzeitig drücken) und starten damit das System neu. Übrigens werden Sie es kaum vermeiden können (ich jedenfalls nicht), im Zuge von Programmentwicklungen in Assembler diese Tastenkombination des öfteren zu benutzen.

Platz beschaffen

Die Diskette ist leider immer noch sehr voll, der Assembler mit Zubehör paßt da nicht mehr hinauf. Bei HiSoft (DEVPAC) ist das Problem wunderbar gelöst (siehe Handbuch), ansonsten müssen Sie wohl für Platz sorgen. Gehen Sie in C-Directory (CD :c) und geben Sie DIR. Sie finden hier alle CLI-Befehle, von denen Sie fast alle löschen können (DELETE Name). Wirklich oft braucht man nur CD, COPY, DELETE, DIR, MAKEDIR und den Editor ED sowie EXECUTE, wenn Sie mit dem Metacomco-Assembler arbeiten. Falls etwas fehlt, kann man ja immer noch die ungekürzte Disk nehmen, oder das Programm wieder hinzukopieren.

Ansonsten hangeln Sie sich ruhig einmal durch alle Directories. Mit »CD Name« erreichen Sie ein »Dir«. Befindet sich im »Dir« noch ein »Dir«, hilft wieder CD. Mit »CD /« kommen Sie eine Ebene höher, mit zwei »/« zwei Ebenen. Mit »CD :« sind Sie wieder ganz oben, nur CD zeigt an, wo Sie gerade sind.

Auf dieser Wanderung können Sie alle Fonts löschen (der System-Font im ROM oder Kickstart-RAM ist nicht dargestellt). Falls Sie kein Bridge-Board oder Sidecar einsetzen, sind natürlich alle Files im PC-Ordner überflüssig. Solange Sie nur einen Drukker haben, brauchen Sie auch nur dessen Treiber (siehe Namen in Preferences), alle anderen löschen Sie. Von den Map-Files (Tastatur-Treiber) brauchen Sie nur d (deutsch) und usa0, falls Sie das DECPAC einsetzen. Übrigens: Meine DEVPAC-Version funktioniert mit der deutschen Tastatur des Amiga 2000 (und 500) nicht. Ich muß vor dem Aufruf von GenAm immer »setmap usa0« geben und dann die Sonderzeichen auf der Tastatur suchen (nach einer Weile weiß man, wo was liegt). Als Empfehlung (ganz sicher) diese Anweisungen:

```
cd :
delete fonsts all
delete utilities all  ;Einen evtl. Error ignorieren
delete #?i.info
delete empty all
delete trashcan all
delete clock
```

Assembler-Tools kopieren

Nun sollten Sie Ihren Assembler und wenn nötig den Linker (ALINK) sowie den Debugger (MonAm) bei HiSoft in das C-Directory kopieren. C-Directory deshalb, weil der Amiga alle Programm-Files zuerst im aktuellen Directory sucht und dann im C. Vorteil dieser Methode: Wo Sie auch sind, die Programme werden immer gefunden.

Nehmen wir an, Sie arbeiten mit ASSEM und ALINK und haben nur einen Drive. Die Include-Files wollen Sie auch; dann sollten Sie so verfahren: Löschen Sie auf der Arbeitskopie die Files wie oben oder mehr und geben dann ein:

```
delete ram:#?
copy c:assign to ram:
copy c:dir to ram:
copy c:cd to ram:
copy c:makedir to ram:
copy c:delete to ram:
assig c: ram:
```

Nun legen Sie die Diskette mit ASSEM und ALINK ein und tippen:

```
cd dfØ:
copy c/assem to ram:
copy c/alink to ram:
makedir ram:include
copy include to ram: all
```

Nun legen Sie wieder die Arbeitsdisk ein und tippen:

```
cd dfØ:
copy ram:assem to c
copy ram:alink to c
makedir include
copy ram:include to include all
```

Include-Tips

Falls Sie nicht alle Include-Files auf der RAM-Disk unterbringen können oder dafür der Platz auf der Zieldiskette fehlt, dann beschränken Sie sich auf die wichtigsten (DOS, Exec, Graphics und Intuition). Aber Vorsicht: Einzelne Files haben die Eigenschaft andere nachzuladen. Gleich zu Anfang des Files finden Sie eine Anweisung der Form »IF NOT DEF Kenner INCLUDE Filename Kenner SET 1«. Das heißt, wenn das Label Kenner nicht definiert ist, dann lade das File und setze das Label. Sehr oft wird allerdings ein ganzer Include-File geladen, um nur ein oder zwei Daten daraus zu verwenden. Da der Include-File auf diese Art auch wieder Include-Files nachlädt, hat man so sehr bald riesige Dateien beieinander.

So wächst dann ein Mini-Programm für den Assembler auf 2000 Zeilen und mehr, was auf Disketten ganz schön Zeit kostet (mit einer Harddisk stört das kaum). In dem Fall sollten Sie die Include-Files maßschneidern. Nehmen Sie den wichtigsten (den Oberfile) xxxlib.i, streichen die Zeilen mit dem »IF NOT DEF« und assemblieren. Der Assembler sagt Ihnen dann, was ihm nun fehlt. Diese Definitionen kopieren Sie nun aus dem an sich zu »includenden« File heraus.

CLI-Tips

Hilfe: Wenn Sie ein Kommando nicht genau kennen, hilft es, das Kommando ohne Parameter zu tippen. Dann sollte »Usage« (die Gebrauchsanleitung) erscheinen. Immer hilft nach dem Befehl ein Blank, gefolgt von einem Fragezeichen.

Mini-Editor

Wenn Sie einmal schnell einen MAKE-File erstellen wollen und nicht für die paar Zeilen den Editor anwerfen wollen, so gilt

```
copy * to filename
```

* ist das Symbol für das aktuelle CLI-Fenster. Beendet wird dieser Modus mit Control-Backslash (\). Mit »copy * to prt:« können Sie auf den Drucker ausgeben, um so Voreinstellungen (über Escape) vorzunehmen oder auch einfach nur für kurze Notizen.

Schneller Kopieren

Im oben genannten Beispiel habe ich über die RAM-Disk kopiert, was sehr viel schneller geht, als über die Disk. Das liegt daran, daß DOS nur einen kleinen Puffer zur Verfügung stellt und so den File in vielen Teilstücken behandelt. Im Fall der RAM-Disk geht es zirka viermal schneller.

Wenn viele Files zu kopieren sind, hilft es, das COPY-Kommando in die RAM-Disk auszulagern. Sonst lädt nämlich DOS nach jedem File COPY neu von der Diskette. Also schreiben Sie vorher:

```
copy c:copy to ram:
```

und schreiben dann zum Beispiel:

```
ram:copy Quell-Liste to Ziel
```

Übrigens: Die Listings zu diesem Buch habe ich alle in einem Directory bearbeitet und wenn sie dann liefen, in das Directory »buch« NICHT kopiert. Setzt man nämlich anstatt COPY dann RENAME ein, erreicht man das gleiche sehr viel schneller, und hat dann noch den Vorteil, nicht auch die Quelle löschen zu müssen.

Workbench plus CLI

Wenn Sie erst nach dem Start entscheiden wollen, ob Sie die WB oder das CLI nutzen wollen, können Sie natürlich mit der WB starten und dann später das CLI-ICON anklicken. Eine Alternative wäre diese: Lassen Sie im File Startup-Sequence die letzte Zeile (endcli >nil:) original stehen und schreiben ein neue Zeile dahinter:

```
NEWCLI CON:x/y/b/h/text
```

Für x/y/b/h müssen Sie Zahlen einsetzen und zwar:
x,y: linke, obere Ecke des Fensters
b,h: Breite und Höhe des Fensters
Für »text«: ein beliebiger Text (der in der Breite paßt)

Das Fenster würde ich zu Anfang nach außen legen und es klein lassen. Bei Bedarf kann man es immer noch mit der Maus verschieben und in der Größe ändern. Auf jeden Fall brauchen Sie jetzt nur im Fenster zu klicken und sind sofort im CLI.

Schnelles CLI

Alle CLI-Kommandos sind Programme, die bei jedem Aufruf von der Diskette geladen werden. Abhilfe bringt, die wichtigsten Kommandos in die RAM-Disk zu bringen (oder alle). Dies geschieht mit

```
COPY kommandoname to ram:c
```

Nun haben Sie zwei Möglichkeiten: Sie setzen jetzt immer vor das Kommando ein ram: oder Sie sagen dem DOS einmalig, es soll das Kommando auf der RAM-Disk suchen. Letzteres geschieht durch

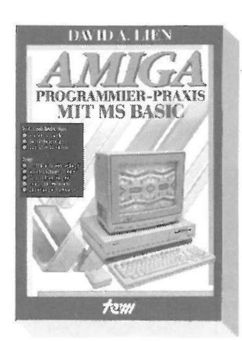
```
assign c: ram:
```

Die entsprechenden Kommandos können Sie auch in die Startup-Sequence schreiben, dann haben Sie den Zustand automatisch nach dem Einschalten.

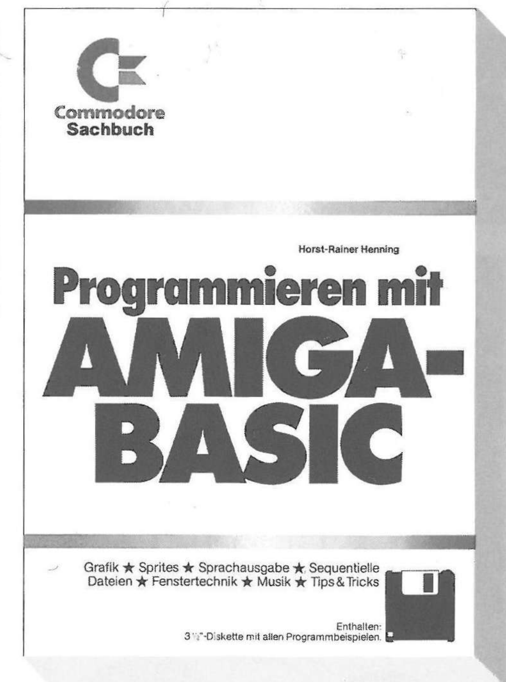
Übrigens: Wenn Sie die RAM-Disk auch unter der Workbench nutzen wollen, brauchen Sie dafür ein Disketten-Icon. Das generiert (auch in Startup-sequence) der schlichte Befehl »dir ram:«.

Stichwortverzeichnis

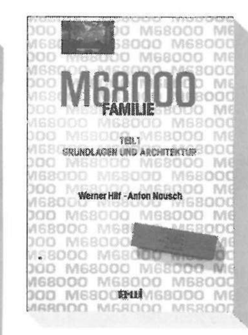
Bücher zum Amiga



Prof. D. A. Lien
Amiga: Programmier-Praxis mit MS BASIC
1986, 400 Seiten
Bestseller! Eine systematische und lebendige Einführung in MS BASIC unter der komfortablen Mausbedienung und Fensteroberfläche des Amiga. Mit über 60 Musterprogrammen zu den Befehlen. Zeigt Amiga-typische Anwendungen: bewegte/farbige Grafiken; Musik- und Sprachausgabe, Strings, Felder, Mathematik, Dateibehandlung, Ein-/Ausgabe sowie »Entwurf von Programmen«.
Best.-Nr. 80369
ISBN 3-921803-69-1
DM 59,–



H.-R. Henning
Programmieren mit Amiga-BASIC
1987, ca. 360 Seiten, inkl. Diskette
Einführung in die Programmierung des Amiga-BASIC: Grafik, Sprites, Sprachausgabe, sequentielle Dateien, Fenstertechnik, Musik, Tips und Tricks.
Hard- und Software-Anforderungen: Amiga 500, 1000 oder 2000 mit 512 Kbyte Arbeitsspeicher, gegebenenfalls ein grafikfähiger Matrixdrucker und ein Joystick, Amiga-BASIC von Microsoft
Best.-Nr. 90434,
ISBN 3-89090-434-3
DM 59,–



W. Hilf/A. Nausch
M68000-Familie: Teil 1 Grundlagen und Architektur
1984, 568 Seiten
Ausbildungs- und Entwicklungstext mit allen notwendigen Informationen über den M68000.
Best.-Nr. 80316
ISBN 3-921803-16-0
DM 79,–

W. Hilf/A. Nausch
M68000-Familie: Teil 2 Anwendungen und 68000-Bausteine
1985, 400 Seiten
In vielen Programmierbeispielen liefert dieses Buch die Praxis der in Teil 1 vermittelten Theorie.
Best.-Nr. 80330
ISBN 3-921803-30-6
DM 69,–



707327

Markt&Technik Verlag AG, Buchverlag, Hans-Pinsel-Straße 2, 8013 Haar bei München, Telefon (089) 4613-0